Profiling and debugging by efficient tracing of hybrid multi-threaded HPC applications

Jean-Baptiste Besnard

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

THESIS SUBMITTED TO THE UNIVERSITY VERSAILLES SAINT-QUENTIN EN YVELINES

Specialised in

Computer Science

at the École doctorale des Sciences et Technologies de Versailles (STV)

in total fulfillment of the requirements for the award of

DOCTOR OF PHILOSOPHY

entitled

Profiling and debugging by efficient tracing of hybrid multi-threaded HPC applications.

---

by **Jean-Baptiste Besnard**

hosted by

CEA, DAM, DIF
F-91297 ARPAJON FRANCE
Département des Sciences de la Simulation et de l'Information (DSSI)

Publicly defended the 16th of July 2014
in front of the following doctoral Committee:

Pr. Alfredo GOLDMAN	Professor at the University of São Paulo	Jury President
Pr. Allen MALONY	Professor at the University of Oregon	Referee
Pr. Michael KRAJECKY	Professor at the University of Reims	Referee
Pr. William JALBY	Director of research, University of Versailles	Examiner
Dr. Marc PÉRACHE	Tutor, Research Engineer at CEA,DAM	Examiner

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN EN YVELINES

Spécialité

Informatique

à l'École doctorale des Sciences et Technologies de Versailles (STV)

présentée pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ DE VERSAILLES

et intitulée

Profilage et débogage par prise de traces efficaces d'applications hybrides multi-threadées HPC.

---

par **Jean-Baptiste Besnard**

Organisme d'accueil :

CEA, DAM, DIF
F-91297 ARPAJON FRANCE
Département des Sciences de la Simulation et de l'Information (DSSI)

Soutenue publiquement le 16 Juillet 2014
devant le jury composé de :

Pr. Alfredo GOLDMAN	Professeur à l'université de São Paulo	Président du jury
Pr. Allen MALONY	Professeur à l'université d'Oregon	Rapporteur
Pr. Michael KRAJECKY	Professeur à l'université de Reims	Rapporteur
Pr. William JALBY	Directeur de Recherche, université de Versailles	Examinateur
Dr. Marc PÉRACHE	Tuteur, Ingénieur de recherche au CEA,DAM	Examinateur

Abstract

Supercomputers' evolution is at the source of both hardware and software challenges. In the quest for the highest computing power, the interdependence in-between simulation components is becoming more and more impacting, requiring new approaches. This thesis is focused on the software development aspect and particularly on the observation of parallel software when being run on several thousand cores. This observation aims at providing developers with the necessary feedback when running a program on an execution substrate which has not been modeled yet because of its complexity. In this purpose, we firstly introduce the development process from a global point of view, before describing developer tools and related work. In a second time, we present our contribution which consists in a trace based profiling and debugging tool and its evolution towards an on-line coupling method which as we will show is more scalable as it overcomes IOs limitations. Our contribution also covers our time-stamp synchronisation algorithm for tracing purposes which relies on a probabilistic approach with quantified error. We also present a tool allowing machine characterisation from the MPI aspect and demonstrate the presence of machine noise for both point to point and collectives, justifying the use of an empirical approach. In summary, this work proposes and motivates an alternative approach to trace based event collection while preserving event granularity and a reduced overhead.

Résumé

L'évolution des supercalculateurs est à la source de défis logiciels et architecturaux. Dans la quête de puissance de calcul, l'interdépendance des éléments du processus de simulation devient de plus en plus impactante et requiert de nouvelles approches. Cette thèse se concentre sur le développement logiciel et particulièrement sur l'observation des programmes parallèles s'exécutant sur des milliers de cœurs. Dans ce but, nous décrivons d'abord le processus de développement de manière globale avant de présenter les outils existants et les travaux associés. Dans un second temps, nous détaillons notre contribution qui consiste d'une part en des outils de débogage et profilage par prise de traces, et d'autre part en leur évolution vers un couplage en ligne qui pallie les limitations d'entrées–sorties. Notre contribution couvre également la synchronisation des horloges pour la prise de traces avec la présentation d'un algorithme de synchronisation probabiliste dont nous avons quantifié l'erreur. En outre, nous décrivons un outil de caractérisation machine qui couvre l'aspect MPI. Un tel outil met en évidence la présence de bruit aussi bien sur les communications de type point-à-point que de type collective. Enfin, nous proposons et motivons une alternative à la collecte d'événements par prise de traces tout en préservant la granularité des événements et un impact réduit sur les performances, tant sur le volet utilisation CPU que sur les entrées–sorties.

Acknowledgement

These words end a three years adventure during which I met fascinating personalities and learned more than ever. First of all I want to thank Marc Pérache my industrial tutor who made everything possible, I hope this work matches the thrust he has placed in me. By its careful and constant guiding he helped me to learn from my mistakes without ever constraining my work or ideas in any manner. This remarkable freedom is at the core of this work as it allowed the springing of new ideas which became my contributions to the the performance tool field. This is this inspiring boldness which is required in the HPC context where legacy is mixed with novelty. I want to thank the CEA/DAM and my hierarchy who allowed me to work at largest scales on world class supercomputers without any form of limitation – allowing me to validate my work on representative cases. I also want to thank William Jalby who directed my work from an academic point of view. Thanks to both his original point of view and long experience in computer hardware architecture he opened new ways of thinking of my problematics through shrewd remarks. I am also grateful that he allowed me to join his team in order to pursue my work on parallel tools. During these years I met several colleagues and friends from CEA and university. I met trainees, doctoral candidates and post-doc from various horizons involved in every stage of the simulation chain: Marc W., Emmanuel, Emmanuel O., Bertrand, Alexandra, Alexandre, Xavier, Thomas, Nicolas, Jordan, Asma and so many more. This was an inciting interdisciplinary environment where we sharpened our point of views, shared good moments and supported each other. I want to thank particularly all the members of the MPC Team, Marc of course, Patrick, Julien J., Jean-Yves, Sébastien, Jérôme, Camille, Augustin, Aurèle, Sylvain, Julien A., Antoine, Emmanuelle and François for everything we shared during this adventure. I also want to thank my reviewers, Professor Krajecky and Malony who took the time to assess my work with interesting remarks in the light of their long experience. I'am also honored that Professor Goldman has accepted to preside my Jury. I want to express my gratitude to my family for its patience, particularly my wife Anaïs for her continous support. Eventually, as I believe that the transitioning to Exascale has just started, I am happy to be able to pursue my work in the HPC field at Paratools, for this I want to thank Mr Shende and Malony.

Contents

CHAPTER 1

Introduction

As numerical simulation is becoming an important tool for scientific competitiveness with various applications ranging from fundamental science (Quantum Physics) to industry (Aeronautic and automotive design), simulation codes have to be seen from the modelling process macroscopic point of view. When dealing with high-end supercomputers, simulation programs cannot be viewed as a tool which punctually validates an hypothesis. On the contrary, the program becomes a constituting part of the simulation process which is aimed at providing measurements matching an experiment. Therefore, although we often picture physicists in their laboratory, performing precise measurements with bounded error rates, nowadays, science is often seen through the lens of a computer program. Moreover, as we will further develop in this thesis, computer programs have few in common with high level equations which describe an objectified reality, instead, they describe an operational reality with iterative and alternative behaviours. This, while coping with evolving execution substrates.

This thesis acknowledges this context and aims at providing developers with the necessary feedback when using supercomputers. Because of their complexity, supercomputers are somehow unpredictable as parallel interactions and contention on shared resources creates a combinatory number of states, sometimes reached randomly because of freedom degrees in the scheduling. Therefore, despite being formally defined at sequential level, parallel execution is sometimes not predictable from the code alone. In the absence of model, a common approach is then to process empirical measurements in order to observe how a code behaves on the execution substrate, approach that we adopted in this thesis by setting up a measurement and analysis framework. The object of this work is then to explore the possibilities of profiling and debugging for production grade applications (million lines of codes) at supercomputer scale (thousands of cores) in an hybrid context. Where, profiling is the examination of a program's performance on a given execution substrate whereas debugging is more concerned by the correctness of the solution or faulty program state (crashes) investigation. The hybrid aspects comes from one of the parallel execution runtime which is targeted by our analysis: the *Multi-Processor Computing runtime (MPC)*. It is a runtime which combines several parallel programming models over an unified scheduler in order to allow their efficient mixing in purpose of taking advantage of upcoming supercomputer architectures. We will pursue this introduction with a brief presentation of the MPC framework insisting on how it constrained our tool implementation. Then, we define our requirements more formally and provide an outline of the organisation of this document.

1.1 The MultiProcessor Computing Runtime

MPC [Pér06, PJN08] aims at providing an unified framework to run massively parallel applications on clusters of (very) large multi-core numa nodes. It supports MPI 1.3, OpenMP 2.5 and POSIX threads interfaces over an unified runtime which is designed to mix those standards in an efficient way [CPJ10]. One particularity of MPC is that MPI processes are running within user-level threads, allowing fine grained scheduling and optimisations such as busy-waiting removal. MPC supports Infiniband and TCP networks with a fully MPI thread multiple support. It is built to allow communication overlapping [DCPJ12] and reduces the overall memory consumption [PCJ09] by factorising process level resources thanks to extended thread local storage [CPJ11] and efficient memory management directives [TCP12]. It is shipped with a patched GCC compiler, allowing both compilation of OpenMP programs and automatic privatisation of global variables in purpose of eventually running program within user-level threads. It also features a patched GDB, allowing transparent debugging of user-level threads [PCJ10] and has been supported in commercial debuggers such as DDT. MPC has evolved drastically since 2006 and is under constant evolution to alleviate the upcoming challenges of many-core architectures. Being used in production on the Tera 100 supercomputer, it reached the petaflopic scale with a competitive memory footprint and a reduced launch time [PCDJ12]. These gains come from the thread based nature of MPC which allows both resource factorisation and reduces the number of MPI processes to launch by a factor which is equal to the number of cores per node (32 on tera 100). Consequently, MPC requires only 4370 processes instead of 140 000 for the whole Tera 100 computer, therefore, drastically reducing launch time[1]. Figure 1.1 presents the Multi-Processor Computing Runtime architecture. As it is a thread based MPI, ranks which are commonly located in distinct processes, are now in user-level threads. This configuration, reduces memory requirements and restores fairness in-between threads which can run on the same scheduler. This avoids for example busy waiting and opens opportunities when mixing programming models.

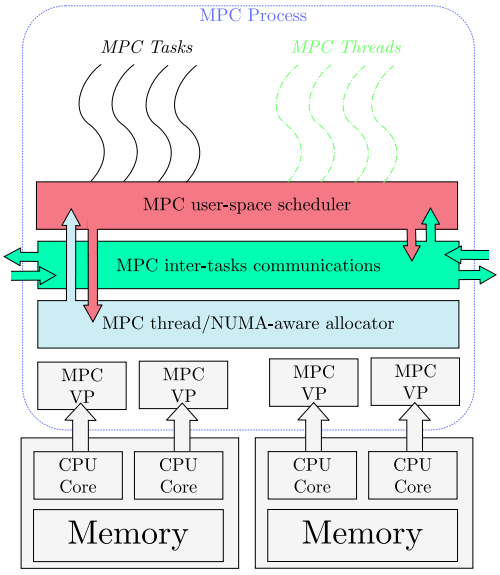


Figure 1.1: *Overview of the MPC runtime.*

[1] 5000 MPI processes is a classical payload on Tera 100 whereas 140 000 is not.

From an instrumentation point of view, supporting MPC has some challenging aspects as it requires a careful handling of parallelism (at task level) and extend the classical process hierarchy (Cluster $\mapsto$ Node $\mapsto$ Process(Ranks) $\mapsto$ Thread) to include the task level (Cluster $\mapsto$ Node $\mapsto$ Process $\mapsto$ Tasks(Ranks) $\mapsto$ Thread), requiring a particular attention when handling program state. Those, parallelism and topology requirements led to the development of our own trace format (chapter 9), allowing, in complement of MPI programs, MPC programs instrumentation which as we will further develop was not practical with existing formats. Consequently, our support of the MPC framework yielded a certain number of requirements which led to the design of our first implementation of a trace analysis framework called the MPC Trace Library (chapter 9). In a second time, these requirements were relaxed when our tool moved to an on-line approach to become MALP (chapter 10), work which has been published in an article [BPJ13].

1.2 Requirements

As described in Chapter 3, managing simulation programs is a challenging task as it adds the classical difficulties of programming with the complexity of modelling. Moreover, the transition toward Exascale which is expected around 2015–2020 [MSDS93] (see figure 2.1) will be more than an hardware problem. Indeed, the increasing number of core per processor (since $\approx$ 2002) already impacted programs as they now have to rely on hybrid approaches. But, current trends, paving the way to Exascale by favouring many-cores and accelerators pose new problematics as they require a dramatic shift in programs architecture — causing close to a complete rewriting of simulation codes. For example the Tianhe-II supercomputer which is at the moment the largest supercomputer includes Intel Xeon Phi for a total of 3,120,000 cores only achieves an efficiency of 62.3 %[2] on the Linpack Benchmark [DLP03], emphasising the arduousness associated with the programming of such demanding architectures. Consequently, porting simulations programs to next generation machines will be a challenging task which requires (1) the capacity of instilling transition by questioning local maximums (or stratification) and (2) the availability of means of measure and control to guide developer teams in the maze of Exascale simulation. This thesis is focused on a small subset of this problem: *measure*. Our purpose is to provide developers with metrics of their programs in purpose of guiding their choice in-between design alternatives (trial & error). In complement, our tools shall be able to *continuously* qualify programs fitting relatively to performance criteria to build a *management metric* — more likely to positively influence developer teams. Dealing with reliability, we also have to explore solutions to describe faulty program states in the context of production jobs (particularly long running batch job) and unpredictable crashes which can be hard to diagnose. Requirements which conducted this thesis to explore both profiling and debugging aspects over a common tracing framework while developing corollary notions such as time-stamp synchronisation and performance modelling.

1.3 Manuscript Outline

In Part I, we first present in more detail the context of this thesis in terms of supercomputer evolution, architecture and their associated programming models. Then, Chapter 3 contextualises the development task in terms of classical management processes (development cycle),

[2] Tera 100 has an efficiency of 83.6 %.

outlining the recursive roles and duties of each of its actors. Followingly, Chapter 4 introduces the role of performance and debugging tools in the light of this development cycle. In Part II, we begin by a brief description of developer tools' architecture before detailing in Chapter 6 work related to our subject. In part III, our contribution starts by presenting our machine characterisation tool, called "MPI Bench" (Chapter 7). It describes which performance can be expected from a given machine from the MPI point of view, measures which can be used by developers to privilege most scalable MPI calls. Then, Chapter 8 introduces the principle of our clock synchronisation algorithm which is needed to restore time coherence within distributed measurement, opening opportunities for time-based analysis. Then, we present the two main parts of our contribution which are associated with two different data management methods: trace-based (chapter 9) and on-line coupling (chapter 10). The trace-based approach is described and contrasted with existing trace formats while introducing its support for debugging. Then, the on-line approach is described as a more efficient coupling method which bypasses the IO bottleneck while providing analysis with enhanced parallelism. Analysis which are covered in Chapter 12 through several modules which were, for most of them, ported from the original trace-based approach to the new on-line trace analysis, demonstrating both debugging (back-traces, deadlock detection, ...) and profiling facilities. Eventually, Part IV, sums up our contribution and conclude this manuscript before outlining our future work.

Part I

Context

CHAPTER 2

Thesis Context

This chapter provides some context on High Performance Computing, focusing on supercomputers' architecture and their evolution. We will start by a brief reminder on supercomputers evolution trends with a description of current hardware followed by an introduction of existing programming models. Our purpose is to insist on the complexity arising from parallel computers as it is at the root of usability problems programmers now encounter.

2.1 Supercomputer Evolution Overview

Supercomputers are known for their rapid evolution, such trend can be witnessed thanks to the well known *top500.org* website [MSDS93] which ranks the world largest supercomputers relatively to the top performance obtained over the *Linpack* [Don87] benchmark. This benchmark, performing linear algebra operations characterises the ability of a given supercomputer to solve numerical problems. Supercomputers are then ranked according to two measurements in FLoating-point OPerations per Second or *Flops*: R_{Peak} and R_{Max}. Where R_{Peak} is the cumulative peak performance of processing units as stated by manufacturers and R_{Max} the operation throughput achieved on the Linpack benchmark. Although the *Linpack* benchmark gives some insight on "real-world" problems, it does not describe applications in general [DLP03] as for example they might stress the interconnection network by performing massive IOs or communications. To address Linpack's limitations, a complementary benchmark *graph500.org* [BBK+10] has been proposed, more focused on data management as it solves graph related problems [FGMM06] and yields results in Traversed Edges Per Second (TEPS). Schematically, real HPC applications are somewhere in-between those two extremes as they perform floating point operations (top500) while managing large data-sets (graph500). Despite aforementioned limitations, *top500* is still at the moment the reference source for supercomputer ranking. Figure 2.1 is from the top500 website, it depicts the computing power evolution since 1993 and projects its evolution until 2020. Exponential evolution of computing power is clearly visible with approximately a tenfold increase every tree years. Moreover, looking at projections, Exaflop shall be reached around year 2020 but not without efforts [BBC+08]. Although an exponential computing power growth has been maintained over the years, supercomputers' taxonomy has evolved drastically, starting in the 1960s with the first supercomputer designed by Saymour Cray where architectures relied mainly on specifically tailored vectorial computing units and a small number of processors, until the 1990s where machines with thousands of processors appeared. At this point programming models migrated from intensive vectorisation (mainly in Fortran) to distributed memory paradigm over MPI or PVM in purpose of taking advantage of the computing power provided by interconnected nodes. From

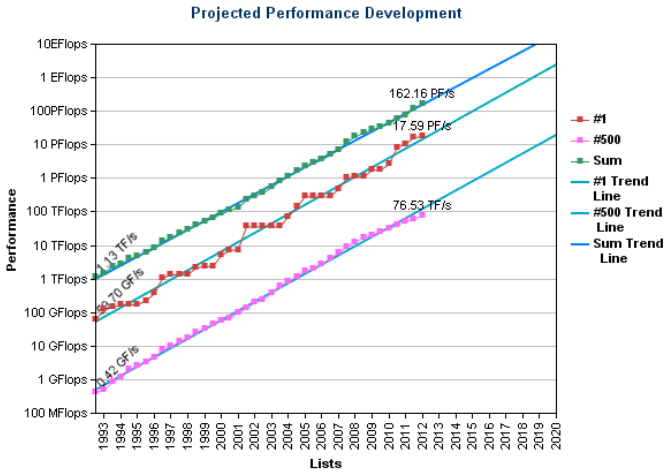


Figure 2.1: *Evolution and projected performance development of supercomputers as displayed on the top500.org website. (Source : top500.org [MSDS93].)*

then until today, most supercomputers started to rely on more standard components (mostly high-end x86 processors) to reduce production costs, MPI took the advantage over PVM and the number of cores started its quick increase until reaching the million with the IBM Sequoia supercomputer and later on several millions with Tianhe-II.

This spectacular increase in the number of cores finds its root in hardware limitations. Until recently, Moore's Law [Moo65] stating the the number of transistors in a given surface doubles every two years has been the main source of computing power improvement as microprocessors were becoming more efficient. Thus, thanks to the increase of processors' frequencies, the 'same work' could be done faster with virtually no application modification. But this trend came to and end around year 2002 when physical limitations (mainly power dissipation) started to slowdown frequency increase, thus, limiting sequential performance gains. Nonetheless, processors manufacturers managed to overcome this limitation, sustaining the exponential performance increase, not in frequency but by multiplying the number of cores. Creating a singularity in code development trends where parallelism became compulsory to get performance improvements — situation which was summed up by the well known "Free Lunch Is Over" [Sut05] quotation.

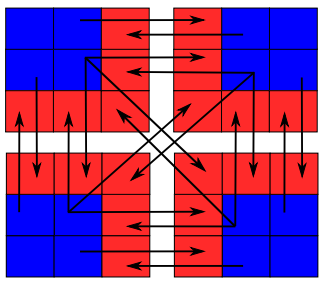


Figure 2.2: *Sample communication scheme for ghost cells synchronisation.*

Because of the rapid increase in the number of cores per node, memory per core decreased and programs were forced to combine both shared and distributed memory parallelism. Indeed, on a given node, using distributed memory parallelism leads to data duplication not only by multiplying processes and their associated file descriptors but also at program level. Figure 2.2 exemplifies this situation with a simple simulation code . It relies on a 2D-mesh and duplicates the same cell up to four times for synchronisation in-between ghost cells (in red) and real cells in blue. This imposed shift to mixed programming requires important program evolution, forcing programs to work at multiple parallelism granularities. In this challenging context, performance tools can provide important feedback to users, helping them to understand and project design choices.

2.2 Supercomputer Architecture and Performance

Since the 1990s, supercomputers gather several computing nodes interconnected by a high performance network. A consequence of cores multiplication, is the hierarchical aspect of processing capabilities. Figure 2.3 presents a simplified assembler code corresponding with a single multiplication (blue box). It can be seen that the operands (A and B) have to be loaded in registers before being processed by the Arithmetic and Logical Unit (ALU, red box) which produces a result which can be stored in the main memory by the "store" instruction. Naturally current processors are much more complex (addressing types, prefetching, branch prediction, ...) and won't be covered in this introduction. But this simple load and store model is sufficient to show that computation is done by combining data (operands/data-set) with data (program). Consequently, it is the memory bandwidth and scattering which "shapes" calculation by defining how programs operate on data-sets within machines memory constraints. Therefore, we start by exposing how data can move in current supercomputers (massive clusters) before exposing some of the programming models being used to exploit them.

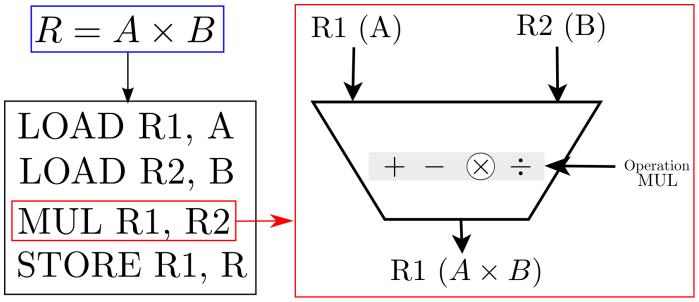


Figure 2.3: *Simple representation of an ALU (Arithmetic and Logical Unit) performing a multiplication on two scalar operands. The left part presents a simplified assembler code performing the multiplication of two scalar values A and B and their storage in R.*

In current supercomputers, multiple data states can be identified both at a given distance in time (or latency) from processing units and with a fixed symbol throughput (or bandwidth). We define a cluster of machines as a group of computing nodes interconnected by a high performance network with a fixed topology. Each node at its turn groups several processing units (or cores each with an ALU) following a topology shaped by their local memory (as known as Non Uniform Memory Access (NUMA) architectures). Context in which some accesses are done to

local memory banks and others to distant ones with variable costs. Figure 2.4 illustrate this trade-off between available memory and bandwidth. It can be seen that when moving closer to processing units, data containers become faster but smaller. Whereas, when moving away from processing units data-stores become larger but slower until reaching file system level.

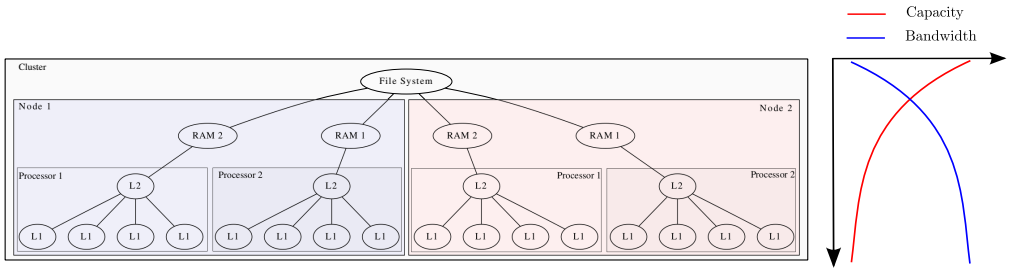


Figure 2.4: *Schematic and effective memory hierarchy.*

As presented in figure 2.5, data can reside in various containers which expose different characteristics, one of the biggest challenge of parallel programming is to actually distribute data according to this topology in order to fully exploit the hardware. Consequently, data have to be distributed evenly among the processing units in chunks which fit into the lowest caches. In this context, two different level of parallelism have to be identified, (1) distributed parallelism in-between the nodes which are connected through the network; (2) Shared memory parallelism which takes place within nodes, where multiple threads access the same memory area. To express parallel computations in these two contexts, several programming models are available each with its own semantic and syntax. Moreover, as aforementioned, memory limitations are forcing programmers to adopt a mixed approach, combining coarse and fine grained parallelism for respectively distributed and shared memory contexts.

2.3 Programming Models

As we developed in previous section, supercomputers parallelism tends to rank memory, complicating memory management and stressing parallel programming models adaptability. Leading to the development of several programming models, each directed towards a particular level of parallelism. This section introduces main approaches, starting by shared memory parallelism approaches, before describing distributed memory parallelism and accelerators.

Data State	Volume	Throughput	Resilient	User Managed
Long Term Storage	Very Large (PB)	Low (less than a GB/s)	yes	no (Storage policy)
File-System	Large (TB)	Low (less than a GB/s)	yes	yes (IO calls)
Remote Memory	Medium (GB)	Medium (network bandwidth)	no	yes (Network calls)
Memory	Medium (GB)	High (GB/s)	no	yes (Allocator calls)
Caches	Low (MB and KB)	Very High (GB/s)	no	no (processor level)

Figure 2.5: *List of common data states in HPC clusters (relatively to a single node).*

2.3.1 Shared Memory

Shared memory is the most common parallelism approach where multiple processing share the same address space, allowing 'direct' data exchanges. Pthread is the most widespread parallel programming interface as it has been standardised in the POSIX standard, becoming basic a block of higher level programming models. Threads are execution streams which can overload the available number of computing units, in other words, operating systems feature a scheduler which role is to switch in-between threads, evenly allocating computing power to each stream. Thanks to this functionality, multitasking is possible even on a single core, using a time-sharing approach. However, as the scheduler is located in the kernel, switching between threads requires a context switch. Therefore, alternatives were developed in order to build user-level threads, able to run multiple threads on top of a single execution stream. Such threads are called user-level threads and can be scheduled very efficiently. Several libraries feature user threads, for example, Marcel [DMN99, Nam01], MPC [PJN08] and GNU portable threads [Eng03]. Relying on threads, several higher level approaches have been developed in order to simplify parallelism expression, featuring various run-times and programming interfaces, ranging from compiler *pragmas* to dedicated programming languages. A common way of parallelising existing programs is OpenMP [DM98] which relies on compiler pragmas, extracting the parallelism from `for` loops while providing a task model. Numerous other parallel run-times have also been developed including StarSS [PBAL09], Kaapi [GBP07] in conjunction with a wide range of programming approaches such as Charm++ [KK93] or Cilk [BJK+95] recursive functions.

2.3.2 Distributed Memory

Dealing with distributed-memory parallelism, the reference programming model is the Message Passing Interface (MPI) [MF08] which relies on messages in-between distributed processes. Several MPI implementations are available including: OpenMPI [GFB+04], MPICH [GLDS96], MVAPICH [KJP08]... MPI generally relies on a combination of both high performance networks and shared memory segments, respectively providing parallelism inside and outside nodes boundaries. One advantage of MPI is its immediate support for NUMA platforms as data replication is enforced by programming model. However, these replications and message buffers overhead, inevitably lead to an increased memory usage. As a consequence, in order to face the rapid increase in terms of number of cores, MPI is often used in conjunction with OpenMP in a mixed programming approach. Limiting the number of processes per node, but, hardening program development. An alternative to message passing is the Partitioned Global Address Space (PGAS) method which consists in splitting memory over threads or distributed processes while providing a transparent access to remote data. Several PGAS implementations are available with for example UPC [EGS06], Chapel [CCZ07], X10 [CGS+05]...

2.3.3 Accelerators

A recent evolution in parallel computing is the advent of accelerators which are complementary devices speeding up computation thanks to a data-flow approach. Such devices generally use the Single Instruction Multiple Data (SIMD) paradigm which applies the same operation to a large data vector. Accelerators were firstly derived from graphic card and programmed through low level graphic calls (shaders), however, the growing popularity of these devices led to the development of dedicated languages, greatly simplifying development: CUDA [Nvi11]

for Nvidia cards and OpenCL [M+09] for both ATI and Nvidia cards. Such devices rely on a large number of simple 'cores', decreasing energy consumption per floating point operation when compared to classical processors. This allowed supercomputers powered with Graphical Processing Units(GPUs) to dominate the Green 500 [SF12] which lists the most power efficient machines. One difficulty of GPUs is that they rely on vendor specific languages, and thus, create an adherence between codes and devices. To face this limitation, several run-times were developed among which are StarPU [ATNW11, Aug11], StarSS [Lab10], HMPP [DBB07] or the recently standardised OpenACC [Ope11] which aims at providing accelerators with a pragma based programming model. More recently, Intel released its Xeon Phi, finding a trade off between GPUs and classical processors by relying on several simple x86 processors (atom like) with extended vectorial operations (AVX) — simplifying code porting but requiring an important optimisation effort.

2.3.4 Summary

This section briefly introduced the variety of approaches which were developed to take advantage of supercomputers and parallel computing units. We have seen that several hardware and software approaches were developed. Some of them being vendor specific and requiring a partial code rewrite associated with a constant optimisation effort. Consequently, parallel programs have to cope with the rapid evolution of both hardware and programming models which necessarily impact simulation codes while requiring a questioning of development habits.

2.4 Thesis Computing Environment

All this thesis measurements were done on two petaflopic range supercomputers. The first one is Tera 100 [TOP10, Vet13](p. 45) (figure 2.6(a)) which belongs to the CEA (Commisariat à l'Énergie Atomique et aux énergies alternatives) which use it for defence applications. The second one is Curie [TOP12] (figure 2.6(b)) which is funded by GENCI and aims at providing the french industrial tissue and research with efficient simulation tools. These two machines are manufactured by Bull SA, featuring a similar designs (see figure 2.7) although Curie uses more recent processors (Sandy Bridge) than Tera 100 (Nehalem), thus, achieving higher performances.

(a) Tera 100 Supercomputer (b) Curie Supercomputer

Figure 2.6: *Views of Tera 100 and Curie supercomputers.*

Characteristic	Tera 100	Curie
R_{peak}	1254.5 TFlop/s	1667.2 TFlop/s
R_{max}	1050.0 TFlop/s	1359.0 TFlop/s
Processor type	Intel Xeon 7500	Intel Xeon E5-2680
Total Number of cores	138368	77184
Total Memory	276736 GB	308736 GB
Memory per core	2 GB	4 GB
Operating System	Linux (Redhat)	Linux (Redhat)
Interconnect	Infiniband QDR	Infiniband QDR
Network Topology	Fat-tree	Fat-tree

Figure 2.7: *Characteristics of both Tera 100 and Curie supercomputers.*

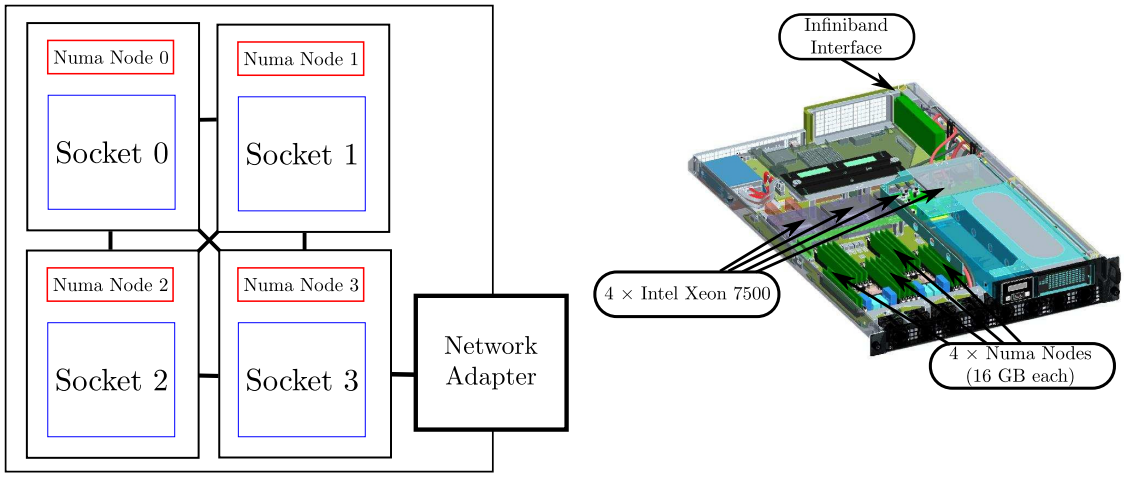
2.4.1 Description

These two supercomputers can be qualified as *generalist* ones as they provide powerful computing units and relatively high volumes of memory per core. They also feature very efficient Inputs and Outputs[1](IOs) and legacy operating systems (Redhat Linux). Other TOP 500 machines adopted different approaches, for example, the IBM BlueGene line, favours a high number of cores with less memory per core (between 512 MB and 1 GB per core) and until recently (BlueGenes moved to linux) specifically tailored operating systems. Therefore, these two generalist machines are able to run moderately parallel payload with acceptable performance, approach not possible with architecture requiring more parallelism because of lower sequential performance. In complement of supporting a wider application spectrum, *generalist* supercomputers have many advantages, for example relatively to legacy codes as they are easier to port. They also have negative consequences as they do not enforce strict parallelism in the development process as such ideal machines somehow maintain the “free lunch” illusion. Moreover, this adaptability requires consequent engineering and administration efforts to hide complexity from the end user, efforts which might not suffice with next generation machines which will feature millions of cores — eventually requiring efforts from end-users. It is already the case at Petascale with supercomputers being less used as tool in a *feed-forward* fashion but be included in the simulation process as an evolving tool which requires trade-offs and *feedback*. Evolution testified by the development of several optimisation tools. Our thesis work acknowledges this context and proposes to develop an optimisation tools which is integrated in the development process, providing constant feedback on program’s performance and positively influencing their adaptation to evolving execution substrates.

2.4.2 Node Description

As presented in figure 2.8(a), a Tera 100 node consists in four NUMA sockets with 16 GB of local memory for a total of 64 GB per node. Each socket hosts an eight core Intel Xeon 7500 processor cadenced at 2.27 GHz, yielding a total of 32 cores with 2 GB of memory per core. Nodes have a single Infiniband [Pfi01, A+01] Quad Data Rate(QDR, 3.2 GB/s) card [2] located nearby a single socket. This configuration creates Non-Uniform Input/Outputs Access (NUIOA) effects [MGN10, Mor11] where one socket has a privileged network access compared

[1] Tera 100 had upon its release a record IO throughput of 500 GB/s.
[2] Each node also feature a Gigabit ethernet interface for administration puposes.



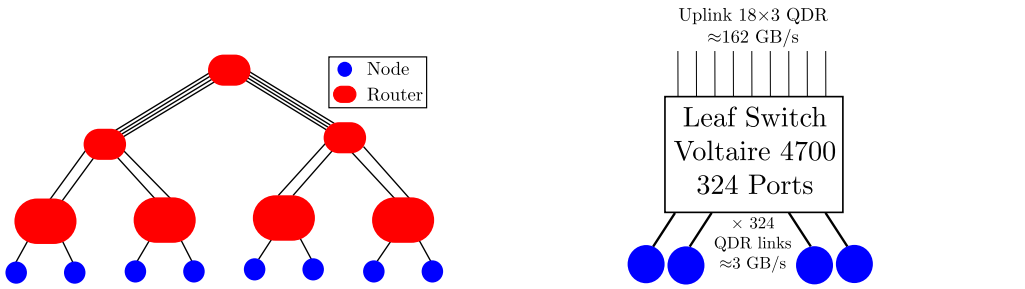
(a) Intra-node topology for the Tera 100 supercomputer.

(b) Actual Node layout for a Bullx S6010 node in 1.5U (3D rendering is from S6010 support page [Bul10]).

Figure 2.8: *Overview of Tera 100 node (Bullx Super-Node 6010) topology.*

to the three others. All these components fit in a Bullx Super-Node 6010 of 1.5 Rack Unit(U). Allowing when L-shaped blade are stacked top-to-tail at a high density of 64 cores in 3U.

2.4.3 Network Topology



(a) Example of fat-tree topology.

(b) Tera 100 compute island network bandwidth.

Figure 2.9: *Overview of Tera 100 node (Bullx Super-Node 6010) topology.*

Tera 100 topology [Vet13](p. 53) is derived from a fat tree [Lei85, Gra03]. As presented in figure 2.9(a), in a fat-tree routers are connected with an increasing number of links which role is to compensate the loss of locality in terms of bandwidth (not in latency as the number of hops increases). Fat-tree topologies also have the propriety of being able to efficiently convey any communication topology with a satisfying efficiency [Lei85]. Propriety which can be understood by looking at bandwidth scattering: thanks to the increasing number of links when climbing the tree, any partition of nodes is guaranteed to reach the full bisection bandwidth. Therefore, different topologies can run over a fat tree with limited performance impact on bandwidth, whereas latency is only impacted logarithmically when the number of nodes in-

creases (tree-based topology). Fat-trees are then interesting topologies for generalist machines which want to be able to run a wide range of codes.

However, if we look more closely at Tera 100 network, it uses a *pruned* version of the fat tree, in order to reduce both network costs and complexity while preserving performances. As presented in figure 2.9(b), computing nodes are regrouped in thirteen *island* of 324 nodes around the same Infiniband router. Therefore, each island gathers $324 \times 4 \times 8 = 10368$ cores in a star topology. Moreover, several island are interconnected with smaller routers (36 ports) to form the whole 140 000 core machine. As depicted in figure 2.9(b), total leaf bandwidth (324×3 = 972 GB/s)[3] is six times greater than the up-link bandwidth (162 GB/s)[4] because of *pruning*. This $\frac{1}{6}$ pruning ratio reduced costs and network complexity while preserving correct performance for regular jobs and full machine runs. More importantly, as most jobs are in the 1000-10000 range, they can take advantage of a regular star topology, while relying on the up-link solely for Inputs and Outputs.

2.5 Summary

This section presented the quickly evolving supercomputing context in which simulation codes which generally evolve at a slower pace have to constantly adapt themselves to new hardware constraints and programming models. We insisted on the topology of supercomputer which has a direct impact over computation scattering and therefore has to be taken into account by developers. We ended this chapter with a presentation of the two machines which are used in the rest of this document. They both feature several cores per node with a non-uniform memory architecture and a high performance Infiniband network relying on a fat-tree network. In the light of this brief introduction, we have seen that these rapidly evolving and complex architectures are very challenging and therefore require a constant effort to be used productively. Effort which we aim at supporting with the tools developed during this thesis.

[3] '3' is approximately the Infiniband QDR (4X) data-rate in GB.
[4] Note that the "service island" has a higher up-link bandwidth of 648 GB/s (216 $\times$ 4X).

CHAPTER 3

Development Cycle

> "**Don't gather requirements — Dig for them**
> Requirements rarely lie on the surface.
> They're buried deep beneath layers of assumptions, misconceptions and politics."
>
> Hunt and Thomas in the Pragmatic Programmer [HT99]
> (Quick Reference Guide)

After we introduced supercomputers and current and upcoming challenges they are associated with, this section briefly introduces software development methodologies which give context to the use of the tools we developed during this thesis. After introducing the purpose of adopting a development methodology, we present classical approaches followed by an outline of several requirements associated with development in complex environment. Eventually, we finish up by exposing advantages which can be provided by tools when being used as heuristics.

3.1 Classical Development Methodologies

A development methodology can be seen as a management approach which aims at coordinating software developers, their managers and clients in a facilitating environment focused on optimising the production of better software. Several organisation models were developed in order to facilitate complex objects conception. Interestingly, all those models feature the same basic blocks, differing more in their usage than in their strict organisation. This section first describes the basic blocks being used in each process from an operative point of view. Then, we present in order of appearance the three classical development cycle.

3.1.1 Constants in the Development Cycle

Development cycle purpose is to formalise the relationship between three entities (1) the client, (2) a development team and (3) the code itself. This first section describes those entities in an empirical context in order to introduce the development process and its goals. To do so, we describe in a methodology agnostic manner some expectations and duties for each of those actors as follows:

- **The client** is at requirements source as he originally formulated them according to his needs. He generally expresses them to potential developers (often through a documen-

tary process) and selects a solution *rationally* (quality of the solution, maintainability, overall costs, ...) suiting his demand after reviewing preliminary design proposals ($Client \leftrightarrow Developer$ iteration). Omitting, intermediate $Developer \leftrightarrow Client$ iteration we are going to analyse from a developer point of view, the client is eventually in charge of judging product quality, attesting that it effectively satisfies his requirements. This last phase, involves, for example, demonstrations, integration tests (...) and eventually leads to the effective deployment at client's site ($Progam \leftrightarrow Client$ iteration).

- **The development team** is central to the development process as it is in charge of (1) understanding the client's need, formulating them as a potential design ($Client \leftrightarrow Developer$ iteration) and (2) expressing this design as a program which fits client's requirements ($Developer \leftrightarrow Program$ iteration). Programmers are therefore an interface between codes and clients, understanding needs and transposing them in a program which exactly[1] fulfils their requirements (generally with design constraints: architectural, costs and design trade-offs, ...). This, while remaining in client's acceptance range.

- **The program** is the development effort final product, supposed to fulfil every requirements while guaranteeing several qualities such as reliability, maintainability, code readability (...). In other words, client's requirements have to be transposed in the program in terms of features ($Progam \leftrightarrow Client$ iteration). However, this process complexity must remain manageable by developers, supposing a known and suitable design ($Progam \leftrightarrow Developer$ iteration).

Development cycle aims at defining interactions between this trinity in purpose of maximising their efficiency. It supposes an ability for each of those entities to maintain a constant *coupling* in purpose of maintaining mutual understanding. For example, the solution space satisfying a given need is generally very wide (choice of languages, definition of the interface, usage patterns, autonomy level, ...). Possibly requiring several client intervention in the design process. Similarly, programmers have to maintain their program in control for example by setting a suitable environment for *monitoring* its features and reliability. Moreover, the development team itself has to be structured (from a management point of view) to face common conception risks (individual cognitive limitations, knowledge dilution, responsibility dilution, ...) and requirements (manageability, planning, productivity, ...). Consequently, it can be seen that the development process involves multiple intricate level of representation with varying constraints. Those levels are connected by several interactions (documentary, oral, formal or not, ...) which can be equally bounded by either communications hazards or phenomenon complexity. From this point of view the development cycle can be seen as an heuristic which codifies interactions in the process leading from a need to a solution. In this purpose, it defines *communication templates*, *efficiency metrics* and *methods*, helping developers to face the complexity of their work.

After this high level development process contextualisation, we will now focus on the main existing models. It shall be noted that such model can be very normative, detailing each document, scheduling interactions, (...). Comparatively, our descriptions will remain brief, as it solely aims at providing a sufficient context to the description of our work in the rest of this

[1] Here *exactly* shall be understood as minimising development costs relatively to classical management metrics (costs, time, workload, risks,...). The development team is supposed to furnish a *rational* solution focused on *satisfying* the requirements, not an ideal solution (see H.A. Simon who develops these notions in [Sim97]).

document. We will try to reference complementary documents providing a wider view on the subject for readers interested in details.

3.1.2 Waterfall Model

The waterfall project management model introduced by Royce [Roy70] is the most "natural" development model anyone would adopt when required to fulfil a requirement. It sequentially goes from gathering requirements to the integration and maintenance of the new product.

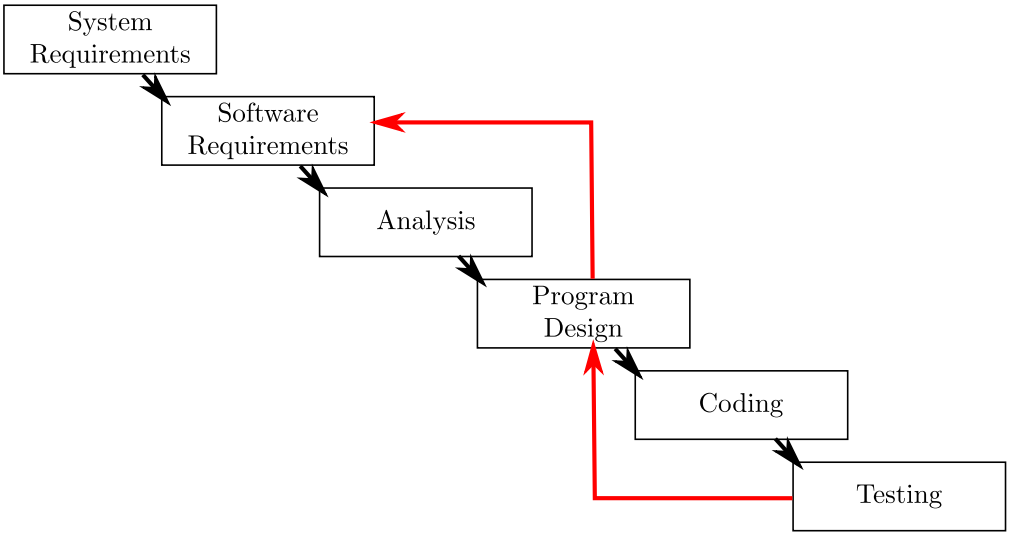


Figure 3.1: *Original waterfall model as introduced by Royce in [Roy70], including feedback loops.*

This process is generally described as a sequential process where phases succeed to each other after a careful validation. In this context, as presented in Figure 3.1, needs and design are both carefully expressed long before the coding phase. This model requires important documentary efforts at each phase, so that clients remain informed of project evolution. Such cycle is also often cited to have the drawback of freezing the design too early in the development process [Mar99, Rot11], neglecting unplanned requirement shifts or unplanned constraints. This effect is called the tunnel effect, after specification phase, clients have "no news" from the developer team until seeing the effective product. This augments the risk of not fulfilling needs because of a communication lack and late risk factor identification as sometimes they can only be identified when implementation begun.

Waterfall was very common in the 80's as it happened to be advocated by large industries [Dep85, Dep88] as the reference model. However, its limitations led to new development approaches which favour a more iterative approach. Nonetheless, it shall be noted that Royce in its original paper [Roy70] already promoted an iterative approach (see Figure 3.1 with feedback loops in red), he even required the waterfall approach to be done twice (p. 7) recognising that some requirement can be sensed only when actually implementing a product. Therefore, even if misapplied as a sequential process, the waterfall model clearly sets the canonical process of a project management model in term of control and reporting but lacks of

adaptability when dealing with the development process itself. Nonetheless, it is still widely used (in modified forms as detailed in the next section), mainly for large industrial or building projects which can reasonably express requirement early in the development cycle. However, software design which is subject to evolving constraints had to rely on iterative methods which eventually led to *agile methods*.

3.1.3 V-Model

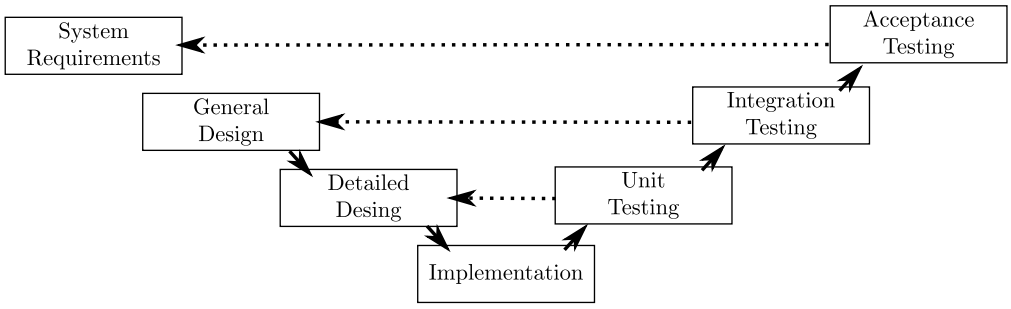


Figure 3.2: *Example of V-Model development cycle.*

The V-Model can be seen as a derivation of the waterfall model where testing phases are designed symmetrically with conception phases. Despite its V shape, this process is still linear and therefore falls in the pitfalls of the linear waterfall model (tunnel effect). Moreover, there are a wide range of interpretation of the V-model featuring different verbosity[2] levels while preserving original waterfall steps. Another ambiguity is in the links between testing and design phases, are they bidirectional or unidirectional ? Again, varying answers can be found. However, V-models are widely adopted for large project management being featured in several methodologies (naturally with variations): the German V-Model [IAB95], United States Department of Transportation guidelines [Uni07], Great Britain Office of Government Commerce PRINCE2 methodology [oGC02]...

As a consequence, the V-Model can be described as an advantaging method in terms of outsourcing as from a management point of view, specification and integration phases are covered. However, dealing with implementation phase itself, it is generally depicted as a single step, the bottom of the V, as far as possible from clients. On implementation side, methods are always iterative as developers progressively fulfil requirements. They might have new interrogations as they get a better understanding of the project, but how does the client answer them in a V process ? Implementing a software project is being able to both understand and fulfil clients' needs. This supposes that the client is able to *express* (specifications) and *judge* (acceptance) the product while developers *understand* (design) and *satisfy* (integration) requirements. In the V-Model this coupling happens only once (and leads to the V shape), whereas, developers and clients might need a stronger coupling to face evolving needs or constraints, observation which led to an iterative development process which aims at preventing tunnel effects by enforcing communication.

[2] An image search of 'V-Model' on any search engine can illustrate this variety.

3.1.4 Agile Methods

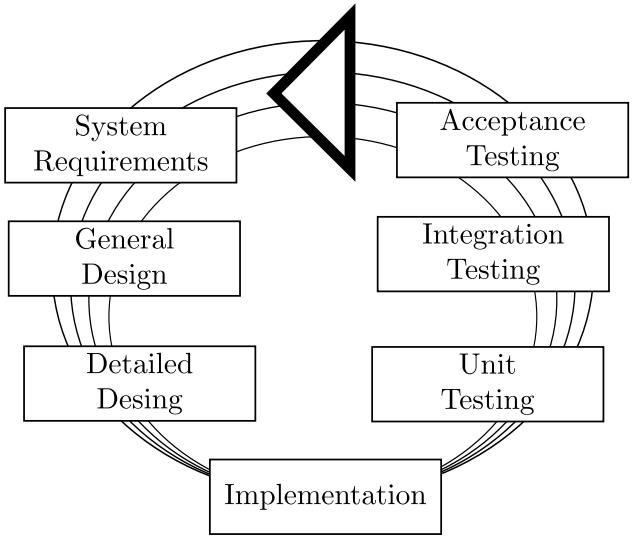


Figure 3.3: *Example of agile development cycle.*

In opposition with classical project methodologies we developed in previous paragraphs, agile methods are profoundly iterative. Their purpose is to prevent tunnel effect by involving clients in development phases in order to confirm requirements and diagnose possible risks early in the development. As presented in Figure 3.3, this process can be seen as an iterative V-process which purpose is to couple client and developers through *control* (requirements) and *measure* (tests) (similar to the Wiener feedback loop of Figure 3.6). Agile methods as introduced in the agile manifesto [BBvB+01] feature a lightweight project methodology, reducing documentary process to a minimum while privileging communication. In that sense, they are more focused on the actual development process than on managing the outsourcing process. For example, developers are able to iterate on technical aspects (for example through intermediate versions) before delivering the final product — guaranteeing client needs are actually understood and satisfied [Rot11].

"**Individuals and interactions** over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more."

Figure 3.4: *Four main values quoted from the agile manifesto as found on [BBvB+01].*

The agile manifesto [BBvB+01], which is quoted in Figure 3.4 insists on four main values which gave origin to a wide range of agile programming models with varying methods: Adaptive Software Development (ASD) [Hig00], Crystal Clear [Coc04], Dynamic Software Development Method (DSDM) [Con08], Rapid Application Development (RAD) [Mar91],

Lean [PP03, WJR07], Scrum [TN86, SB08], (Rational) Unified Process ((R)UP) [JBR12], eXtreme Programming (XP) [Bec00]. All these methods apply the agile precepts to various industrial processes while remaining focused around the development process and therefore developers. In this purpose several agile methods promote development specific processes such as *continuous integration*, *unit testing*, *pair programming*..., going beyond classical management process by clearly taking developers and by extension programs into account. Consequently, those methods which are becoming the reference in term of software development take advantage of both an holistic and iterative approach which is more suitable to manage complex and evolving projects, even from an industrial point of view [Dep94, Def05].

3.2 Developing Against Complexity

This section proposes to analyse the development process from a systemic point of view in order to emphasise the interactions between each entities. Instead of describing punctual processes or documents (as in previous methods descriptions), we propose to start from a global system layout which will be analysed in terms of interactions.

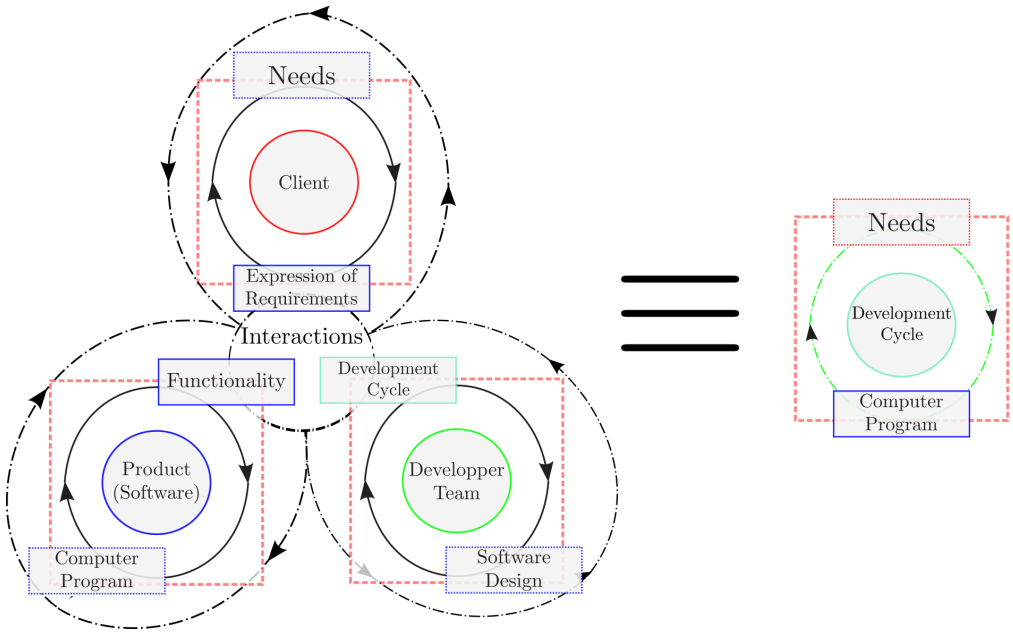


Figure 3.5: *Systemic view of the development process in term of operative systems.*

Figure 3.5 depicts a development process systemic model, as mentioned in previous section it is a tripartite process involving a client, developers and a program. Boxes with plain lines are linked to observable/communicable events, for example *client's requirements* can be expressed as a document detailing its needs. On the contrary, boxes with dashed lines are abstract models which are not directly communicable, thus, requiring a transposition to an observable state (shared representation). Figure 3.5 is designed from a client point of view as it focuses on the shift from a need to a satisfying program. More importantly, from a macroscopic point of view, this description is neither exhaustive nor unique. For example, a program is an observable of a computer system, a design an observable of a program model and a need

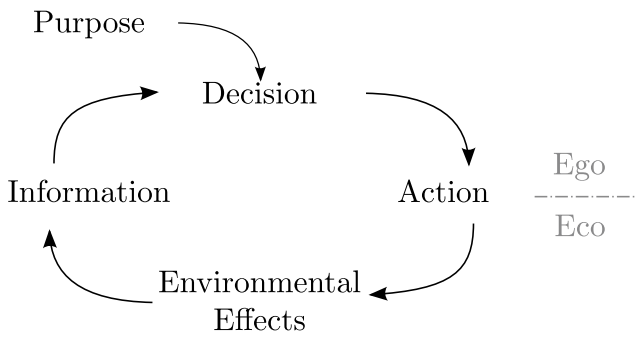


Figure 3.6: *Feedback loop described in [Wie61].*

can be an observable of an organisation. Moreover, roles are not fixed as an observable can become a model (features can be observed from the program, a design possibly describes an abstract program, ...). Starting from Figure 3.5 (right) we can see a compact representation of the development cycle which is a substrate allowing the shift from client's need to a computer program with developers (green loop) in charge of guaranteeing this transposition. Looking at the left part of the figure, the process appears in more details with its interactions and abstractions. More particularly, if we analyse interactions, two types of loops can be identified and will be referred to as structural and catalysing ones with three occurrences of each.

Such loops have to be observed as feedback loops (from cybernetic theory [Wie61]). A cybernetic loop (see Figure 3.6) can be defined as an action which observes its effect in order to guide further actions. Notably, such loops are comparable to operative models which associate an empirical measurement with its abstract representation, for example in this case the $\{decision, purpose\}$ couple, models and is modelled by environmental effects. Recursive relationship which applied symmetrically (by separate entities), and considering bounded rationality [Sim97] are studied by *convention theory* [Lew69, BT87, Amb03] which states that cooperative behaviour requires and creates shared representations which cross individual boundaries.

3.2.1 Structural Loops

Structural loops are in charge of coupling entities and are subject to communications related loss as they go through the environment. They generally rely on a shared symbol system (most of the time oral or written language) to convey metrics associated with abstract representations. They have no explicit substrate and therefore have to be maintained as inputs and outputs of entities, being the consequence of a symmetrical effort from those entities. This bipartite aspect makes them relatively fragile as feedback can only be achieved bidirectionally. These three loops can be described as follows: $Client \leftrightarrow Developer$, $Developer \leftrightarrow Program$, $Program \leftrightarrow Client$, each of them associated with a development cycle requirement and possibly modelled as a cybernetic loop (Figure 3.6) which expresses a certain form of trade-off/adaptability.

Figure 3.7 illustrates those structural loops by expressing the exchanges of expectations and internalised duties from the point of view of each entity. Ideally, such loops shall lead to a symmetry between inputs and outputs of entities involved in the development process

Interaction	Example of process
Client → Developer	Did I express my needs ? Did they evolve? Are they well understood by developers ? Are developers within schedule ? Can I attest of their efficiency (cost, quality, ...) ?
Developer → Client	Are Client's needs/requirements realistic, contextualised, detailed enough ? What are our constraints ? What is our proposal and at at which cost ?
Developer → Program	How to express the design in a program which matches requirements ? What is the most efficient scattering of work amongst developers ? Are there known problems, how does this program section work, who is responsible for it ? How to guarantee and attest for reliability and functionality ?
Program → Developer	Is the code easy to develop (dependencies, compilation process, organisation, coding conventions, ...) ? Does it matches the preliminary design, if not why ? It its complexity manageable ? Is every section tested for reliability, performance and functionality (unit, performance and integration testing) ?
Client → Program	Does it satisfies my requirements ? Is it easy to deploy (target machine, unplanned constraints, ...) and maintain (requires external maintenance, licences, extensible to future needs, ...) ? Is it integrated in our processes ?
Program → Client	Does it provides functionality in an actionable fashion (Human Machine Interface, organisation specific processes or constraints, ...) ? Is it transferable to client's computing environment (dependencies, licences, reliability ...) ? Is it possible to guarantee features over time, if not why ?

Figure 3.7: *List of structural loop with examples of expectations / internalised duties.*

(in order to fit in the feedback loop paradigm [Wie61]). Such loops are therefore correlated with the ability to *control* and *measure* external processes. Points which are often defined by development models as a documentary process, testing, meetings, team building...

3.2.2 Catalysing Loops

Catalysing loops can be described as an internalisation process which converts an observation to a decision which satisfies an individual purpose according to relative heuristics. These loops are associated with a substrate (client, developer, program), allowing them to operate separately. Their decision process is optimised to resolve a given set of problems (specialisation) which purpose is to satisfy the needs of their respective organisations as rationally as possible [Sim97]. Three loops can be extracted from Figure 3.5: Needs ↔ Requirements, Program ↔ Functionalities, Design ↔ Development cycle.

It can be seen from Figure 3.8 that catalysing loops have a contextualisation role. Indeed they match the actual development process with abstract models of surrounding organisations in order to ensure a sufficient coupling without having to expose their complexity to every other entities. For example, a program evolves within a computational hardware which imposes its constraint, client's needs are part of its own organisation which requested their expression as a computer program, software design obeys to rules which aim at optimising the development process (UML modelling, component reuse, building process, complexity management, ...). More importantly, the development process finds its roots in the specialisation of models to a particular purpose by discrete agents who by nature have diverging point of view on a given problem (because of their specialisation, sort of "language gap").

Interaction	Example of process
Needs → Requirements	How to express and systematise the needs of my organisation ? Will they evolve ? Does this need fits in the processes of my organisation ?
Requirements → Needs	Are there unplanned constraints, are they acceptable ? Is this expression of requirements actually satisfying my needs ? Were they well understood ?
Program → Functionalities	Does the program efficiently solve its problem ? Are features constrained by the hardware ?
Functionalities → Program	Does it satisfies the requirements ? Does it reflects the design ? Is it possible to guarantee features ?
Design → Development cycle	What are the most efficient processes to design a program ? Are features identified and explicitly separated ? What are the software components needed to satisfy client's need ?
Development cycle → Design	Does design describe a program which satisfies client's requirements within machine constraints ? Are programmers in possessions of a sufficient knowledge to execute their work ?

Figure 3.8: *List of catalysing loop with examples of constraints/trade-offs arising from interaction and respective contexts.*

From a global point of view, all the entities involved in the development cycle are looking for a point of agreement which satisfies a multidimensional problem going beyond individual rationality. Their purpose is to efficiently express constraint arising from their surrounding organisation (computer hardware, client organisation, software design) as a *satisfycing* [Sim97] computer program. In this purpose, they have to *organise* themselves in a process favouring communication while making critical development cycle aspects observable through synthetic metrics. Encouraging the constructive search for a *satisfying* trade-off. Development cycles are in charge of impulsing this process by setting up a *convention* between all the participants, they codify interactions in order to achieve such trade-off. From this point of view, any development process starts from specifications and finishes with a test phase — measuring the achievement of the process. Main differences being in methodologies temporal aspects which evolved concurrently with programs complexity and volatility from a linear process (waterfall) to a fully iterative one (agile methods). Besides, as we detail in next section, in conjunction with this holistic process, development methodologies are more and more relying on tools to improve productivity for both structural and catalysing aspect, naturally integrating them in the development cycle.

3.3 Tools as Heuristics

Tools are now compulsory to develop efficient codes for various purposes ranging from code development to software documentation. This section proposes to list tools which are commonly integrated in the development cycle in terms of functionality. These tools could be viewed as heuristics as they either create, convoy, preserve or inspect information alongside the development cycle — helping actors in their development task.

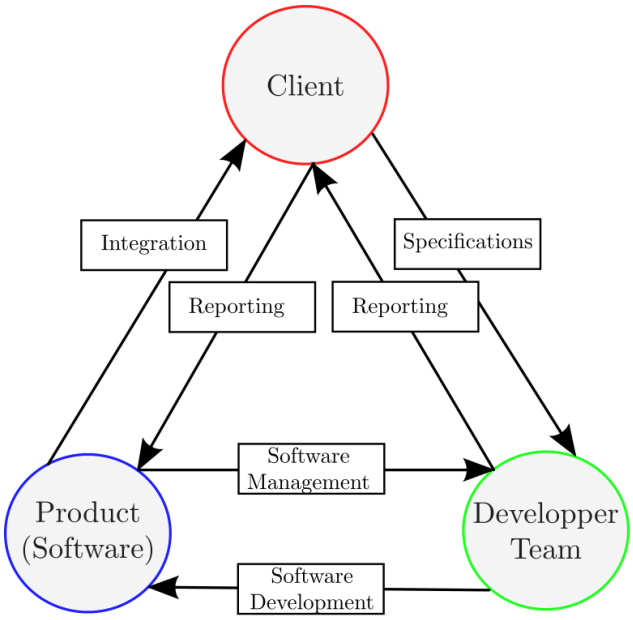


Figure 3.9: *Classes of tools involved in the development cycle.*

As presented in Figure 3.9, six interactions types can be derived from the client, developer, program trinity. Each of them taking advantage of tools or normalised processes to gain in efficiency. We propose to derive five classes of tools from these relationships: reporting, integration, specification, software development and management. This section details each of these classes with sample tools and usage patterns in purpose of contextualising common uses of tools within the development cycle.

3.3.1 Specifications

Specifications are the starting point of any project, they define its purpose and constraints (costs, hardware, ...). In general, their expression is done through a *documentary process* which allows their formal transmission to the development team. However, supplementary communication channels are also used to complete the specification process, for example an initial *meeting* allows to initiate an *interaction process* between the participants in order to contextualise the initial document. Moreover, less formal communications has to be taken into account (e-mail, phone-calls, ...) as an efficient information vector which can possibly outcome conventional organisations, allowing clearer requirements definition.

3.3.2 Software Development

Dealing with software development, a lot of tools are commonly used by developers. At first the *compilation chain* is used to validate the program from a static point of view. As far as the code is concerned, *Integrated Development Environments (IDEs)* can help developer in the management of their code by providing useful features such a syntax highlighting, code completion, templates... *Version control* tools such as git or SVN can be used to handle various versions of the same code, tag specific versions, sharing them between developers. Moreover, on software design side, several *modelling tools* can be used to create UML diagrams depicting program layout.

3.3.3 Integration

Integration can be described as the phases which precede the actual deployment at client's site. It starts by a macroscopic validation through *integration testing* which purpose is to validate global features. Some products also rely on a *phased release*[3] in order to progressively stabilise the product with clients (or in open-sourced development, users) feedback, this has the advantage of providing a realistic test coverage but requires interlocutors who can temporarily put up with unstable software. During this phase, for example with continuous integration, developers and clients can report defects or iterate on features and how their are brought to the user (interface for example) to enhance the product through successive adjustments.

3.3.4 Reporting

Reporting couples clients with programmers, relatively to specifications, project progression, unplanned difficulties, documentation... As far as the client is concerned, he can report his observations, request features or describe defects he encountered on a given program version. This process can take place through various communication channels (see Specifications), although, current trend is to gather all those aspect as a *wiki based portal* which can be accessed by both developers to enrich the knowledge base and clients to report defects or make feature requests (as tickets). Such platforms(github, source-forge, bitbucket, ...) were developed mainly to fulfil open-source requirements for decentralised use and development and start to be used (declined as a commercial products) within the industry in support of the development process.

3.3.5 Software Management

Software management covers all the methodologies which allow programmers to monitor and understand their code in term of reliability and efficiency. *Unit testing* which provides feedback to developers, particularly during refactoring[4] phases as they guarantee individual component features. *Software documentation* focused on technical aspect is also important to face turnover while avoiding knowledge dilution amongst developers. It can be done cooperatively using for example a dedicated *wiki* which is easy to access and update. The particular case of performance tools and debuggers will be discussed in more details in section 4 as it embeds the work of this thesis.

3.3.6 Overview

As we have seen with the list of tools we enumerated, developers are evolving in an environment which is indistinguishable from its tools. Successively providing valuable features (debugger, profiler, editor, ...), giving rhythm to the development cycle (unit-testing, integration tests, planning,...) and conveying communication (source versionning, bug tracking, online documentation, ...). Tools are therefore facilitators in the development cycle as they optimise and carry repetitive tasks, helping and inciting developers to follow development process, transitively finding a trade-off.

[3] Debian for example has three levels: Stable, Unstable, Testing.
[4] Action of changing code design while keeping similar software components.

3.4 Summary

This Chapter started by introducing common development methodologies, insisting on their overall similarity. Then, we proposed a systemic analysis of such process by focusing on interactions. Analysis which highlighted two types of coupling that we ranked in *structural* and *operative* loops. Operative loops have a contextualising role, taking advantage of both individual skills and local constraints. Whereas, structural loop have to convey both requirements (control) and metrics (measure) in order to establish *feedback* in search for a satisfying tradeoff. In a second time, we analysed how tools match this tripartite process by mapping classes of tools over structural loop, emphasising their heuristic aspect. However, as the purpose of this work is to provide some building blocks among many others, the importance of the development process as a whole has to be pointed out as a requirement for quality and productivity. Dealing with this thesis, we focus on the coupling between developers and their program. The increasing complexity of the computing substrate tends to emphasise the empirical aspect of the programming task, forcing developers to rely more and more on trial and error approaches. Consequently, developers need compact performance metrics to asses the quality or badness of their choice on a daily basis, making the uses of tools to both qualify reliability and performance of programs compulsory — tools which are the object of next chapter.

CHAPTER 4

Role of Performance and Debugging Tools

> L'intelligibilité du compliqué se fait par simplification [...].
> L'intelligibilité du complexe se fait par modélisation [...].
>
> Jean-Louis le Moigne in La Modélisation des systèmes complexes [Moi99](p. 10).

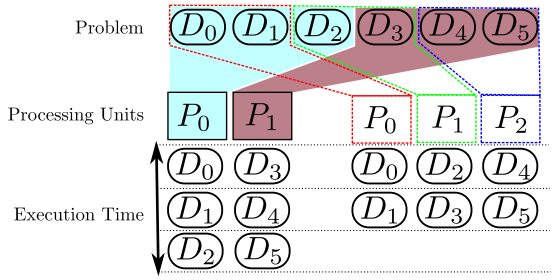
Classes of software management tool which are particularly interesting are are *performance and debugging* tools. Such tools aim at ensuring that programs efficiently use supercomputers' resources. This supposes that a supercomputer can be misused, subject that we will firstly develop from both performance and functional aspects.

4.1 Performance Metrics

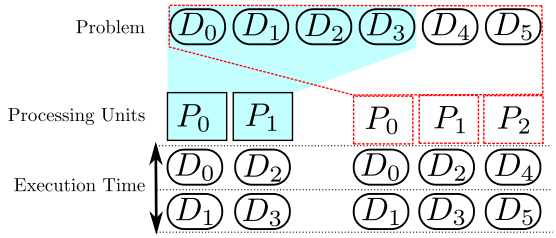
As parallel programs can be inefficient, we first introduce common metrics which objectively define program performance. We start with common metrics as speedup and efficiency and derive the Amdahl law which sets an upper bound to the speedup (assuming strong scalability). Then, we conclude over challenges associated with the need for scaling simulations in the context of current and upcoming supercomputers.

4.1.1 Strong and Weak Scaling

As presented in Figure 4.1, there are two approaches when scaling a given problem on a supercomputer. To begin with, as presented in figure 4.1(a), it can be used to provide the same result in a reduced time frame, allowing a more productive use for example when doing parametric searches. In such case, problem size is kept constant while the number of processing unit grows — approach referred to as *strong scaling*. However, as depicted by Figure 4.1(b), a larger processing power could also be used to process a larger problems, such as the problem size remains constant on each processing unit — approach called *weak scaling*.



(a) Strong scaling.



(b) Weak scaling.

Figure 4.1: *Graphical comparison between strong and weak scaling.*

4.1.2 Canonical Speedup

$$S(n,p) = \frac{\text{Sequential Time}}{\text{Parallel Time}}$$
$$S(n,p) = \frac{seq(n) + par(n)}{seq(n) + \frac{par(n)}{p} + comm(n,p)} \tag{4.1}$$

The *speedup* is commonly defined as the sequential time over the parallel time, yielding the acceleration achieved by the parallel program when compared to its sequential counterpart. As presented in equation 4.1, this quotient can be expressed through three components with p the number of cores and n the problem size:

- **A sequential part** $seq(n)$ which describes time spent in serial sections which cannot be made parallel. Commonly program initialisation, finalisation and input/outputs contribute to this sequential term.
- **A parallel part** $par(n)$ which depicts computation which are distributed among processing units, therefore, divided by p when running in parallel.
- **Parallelism overhead** $comm(n,p)$ which accounts for the supplementary processes required by parallelism such as communication, memory duplication, contention, ... This factor is generally an increasing function of both problem size n and processes count p.

By looking at equation 4.1, which presents the canonical speedup definition. Speedup can be bounded in two ways either by sequential part or because of parallelism overhead. Consequently, performance tools are aimed at identifying and help limiting these two factors in order to achiever higher scalability. Speedup can be expressed in a more compact fashion as

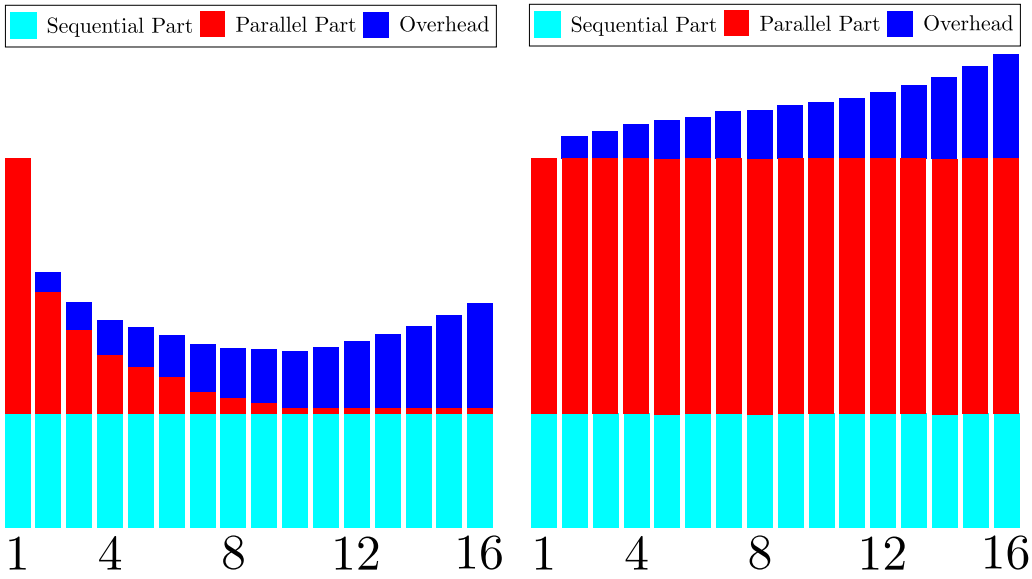
an efficiency which describes how a program achieved to be accelerated by p processing units when compared to an ideal acceleration p, leading to:

$$\epsilon_{acc}(n,p) = \frac{S(n,p)}{p} \tag{4.2}$$

Efficiency as shown in equation 4.2 is a measurement of the achieved acceleration. This metric does not take into account problem growth which would also be qualified of "more efficient". Therefore, such efficiency is only correlated with computation acceleration (strong scaling) when larger problems (weak scaling) are expected to remain at constant speedup (close to constant execution time) yielding a decreasing efficiency (at least in $\frac{1}{p}$, plus parallel overhead) although processing larger problems.

4.1.3 Scaling Bounds

Starting from previous scaling definitions in terms of weak and strong approaches, different kind of bounds can be derived from the execution substrate. This section proposes to observe bounds derived from speedup equation from a more practical point of view.



(a) Strong scaling. (b) Weak scaling.

Figure 4.2: *Execution time for strong and weak scaling scattered among speedup factors.*

As it can be seen in Figure 4.2, weak and strong scaling lead to very different behaviours at scale. As we mentioned before, *strong scaling* aims at *accelerating* the computation by scattering it on an increasing number of processing units. However, as presented in Figure 4.2(a) this process has to face two main limitations: (1) the parallel computing time divided infinitely converges to the sequential time, yielding the classical speedup boundary described by Amdahl's law: $s(n,p) \leq \frac{1}{seq(n)}$. Moreover, (2) parallel overhead which is an increasing function of the

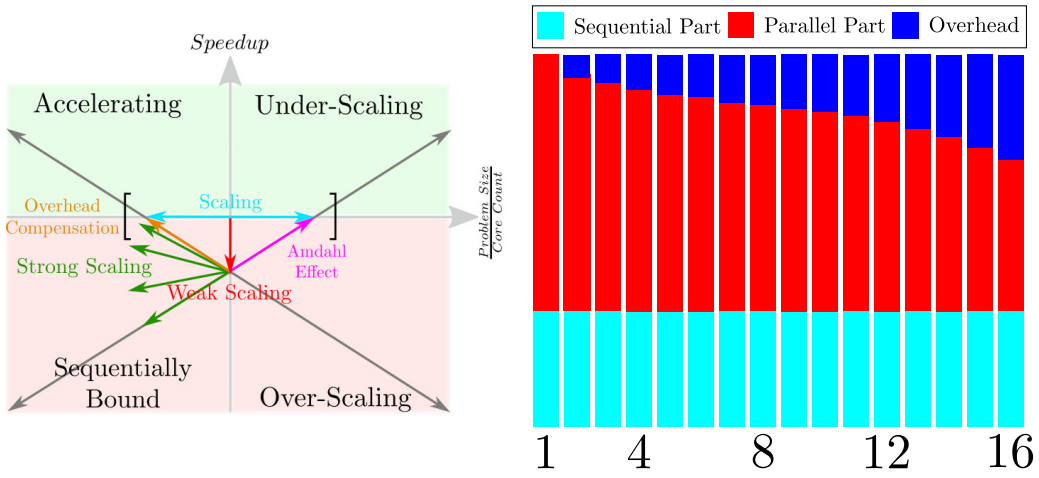
number of cores tends to limit the speedup (in our case at ten processes). Therefore, limiting the sequential part and communications' complexity are two conditions needed to accelerate a program. Moreover, if parallel computation complexity is higher (in function of p the number of processes) than the one of both sequential part and overhead, larger problem sizes are associated with higher speedups (more computation less communications) — observation which is called the *Amdahl effect*. We can conclude that in presence of a sequential computation time and/or communication costs, acceleration is *always* bounded, preventing strong scaling on current and upcoming machines. In other words, a single node cannot hold problems addressed by the whole machine and even if it could, time required to process this problem would be impractical. Besides, when looking at the tendency which promotes a larger number of simpler cores and less memory per core, we can only emphasise the increasing constraints on the *strong scaling approach*: (1) lower sequential problem size and performance, (2) lower problem size increase per core (linked to memory per core) and (3) higher overhead (more distinct computing units). This makes the acceleration of a fixed problem on a whole machine illusory, except if embarrassingly parallel with a data-set which is sufficiently large and does not has to fit in memory (i.e. generated) such as for example in crypt-analysis applications.

Dealing with *weak scaling*, as presented in Figure 4.2(b), it aims at linearly increasing parallel computation time core count in order to keep it constant relatively to a single computation unit. In such case, parallel time remains constant by construction, leveraging Amdahl's speedup limit. Despite, sequential time and overhead are still bounding the speedup. Such context is described by Gustafson-Barsis law [Gus88] which is $S(p) = p - \alpha(p-1)$ with p the number of core and α the sequential fraction parallel processes (including overhead). This equation models an unbounded linear speedup under with a growing problem size. This speedup is referred to as a scaled speedup as it compares the execution time to a sequential problem which cannot be measured experimentally because of time and memory limitations. Looking at the α factor, it depicts the sequential time relatively to a single process, factor which might increase with p for example because of a growing communication cost. Therefore, in order to achieve, "infinite" scaling as described by Gustafson-Barsis law, communication cost (and more generally overhead) shall have a cost of lesser complexity than the computation of a problem of size $O(n)$, bound to grow linearly with the number of cores. More practically, communication cost is generally bounding weak scaling, yielding the behaviour of Figure 4.2(b). Where computation time increases with the number of core because of a growing overhead, eventually preventing programs to solve a linearly growing problem in a fixed time. Therefore, the main limitation to *weak scaling* are the sequential fraction and the overhead which have to grow at most linearly in function of p, remaining therefore constant per processing unit — allowing scaling in a fixed time frame.

4.1.4 Acceleration versus Scaling

This section analyses how scaling is related to acceleration. As common measurements such as *weak* and *strong* scaling are somehow meta-concepts which as we mentioned before are linked to a wide range of variables such as problem size, communication complexity, sequential part ... We propose to sum up all those factors in a graphical fashion, making those concepts more actionable while emphasising programs behaviour relatively to speedup.

As presented in Figure 4.3(a), we propose to analyse the relationship between speedup and problem size per core in order to outline and sum up the concepts we developed. In this fig-



(a) Speedup in function of problem size per core. Vectors represent possible evolutions for a growing number of computing units.

(b) Computation time when scaling in presence of increasing communication cost requiring problem size compensation.

Figure 4.3: *Illustration of speedup and scaling factors in term of problem size per core.*

ure, we consider that computation over a linearly growing data-set has an higher complexity than the overhead, which mainly accounts for communication costs. Besides, we also suppose that computation cannot be scattered symbolically (for example as in key space exploration) but instead relies on a distributed data structure both growing with the number of cores and requiring communications for spatial coupling[1]. *Weak scaling* keeps problem size per core constant but faces an increasing overhead when increasing core count, making of "weak-scaling speedup" (red vector) a decreasing function of the number of cores. This, because of an increasing overhead when compared to the work per core (see figure 4.2(b)). Consequently, in order to scale problem (cyan vector) in constant time as in Figure 4.3(b), problem size might have to be decreased in order to compensate overhead (orange vector). However, considering that overhead has a lower complexity than computation, its ratio can be diminished by increasing problem size — effect commonly called the *Amdahl effect* (magenta vector). Effect bounded by memory per core, naturally limiting problem size (represented as brackets on the $\frac{\text{Problem Size}}{\text{Core Count}}$ axis). Dealing with strong scaling (green vectors), its gains also depend on both overhead/computation complexity and maximum sequential problem size. When infinitely dividing a problem over a growing number of cores, the program fist accelerates but fatally comes a moment where the potential computational gain is lower than the associated overhead cost — the program being sequentially bound. In summary, Figure 4.3(a) proposes four regions describing the *speedup-scaling vector*:

- **Accelerating region:** in this region code runs faster on a larger number of cores. This corresponds to a decreasing computational cost per cores and therefore a lower problem size per cores. Cost which has to decrease sufficiently to compensate parallel overhead. Outlined by a decreasing speedup when doing weak scaling (see figure 4.2(b)).

- **Under-scaling region:** if problem size grows less than linearly with the number of

[1] This is the case of most simulation codes which rely on a spatial decomposition and communications at each time-step.

cores p, problem size per core decreases with p, leading to an increasing speedup if the computational gain compensates overhead.

- **Over-scaling region:** if problem size increases at least linearly with the number of cores p, problem size per core either remains constant or augment with p. Necessarily leading to a decreasing speedup respectively because of parallel overhead (weak scaling) and increasing problem size.

- **Sequentially-bound region:** when problem size per core is too small compared to parallel overhead or sequential computation, a larger number of cores and therefore further dividing the problem leads to a decreasing speedup, degrading program performances.

Dealing with scaling a problem on a larger number of cores, we already mentioned that *strong scaling* is illusory for domain decomposition problems because of both the limited sequential problem size and parallel overhead which bounds speedup. However, when coming to *weak scaling*, we shown that it is also impacted by parallel overhead, yielding a decreasing speedup in function of core count. In every cases, this overhead has to be compensated when scaling either by growing problems (Amdahl effect) but more generally – as they already fill computing units' memory – by reducing problem size. Thus, when scaling, the linear problem size growth P_s on p cores could be denoted $P_s(p) = s.p$ with s the size per core. However, to run in constant time, this problem size must compensate parallel overhead, being reduced by a size matching the overhead (in time) in terms of computation. If we denote $C(n)$ the sequential computation time for a problem size of n, we need to find, n_{comp} a compensation size such as $C(n_{comp}) = Parallel\ Overhead$. This yields, $P_s(p) = s.p - n_{comp}$ and makes higher computation costs preferable than lower ones as they reduce n_{comp}. Moreover, in order to allow increasing problem sizes, compensation size n_{comp} shall be of lesser complexity than the problem size itself which is in $\Theta(s.p)$. In other words, compensation size have to be at most linear, yielding $P_s(p) = (s - s_{comp}).p$ with s_{comp} the compensation size per core, being as a result a constant which compensates a constant overhead $C(s_{comp})$ linked to computational complexity. Consequently, in order to scale without limitation on a large number of cores, programs have to keep their overhead independent from p and more practically as far as communications are concerned to communicate with a constant number of neighbours. For example, collective communications with a growing number of cores have to be avoided as much as possible as their cost per core for the less expensive ones grows in $O(ln(p))$. Cost which even if logarithmic will eventually limit problem size at larger scales.

4.1.5 Summary

We have seen that several factors prevent programs from scaling on supercomputers. Relatively to maximum acceleration we outlined that it is bounded by the sequential part as expressed by Amdahl's law. We also emphasised that strong scaling is not feasible at supercomputer scale except for a limited range of problems featuring symbolic data-sets or expensive computation [2]. On the contrary, in order to scale a program supercomputer-wide, problem also has to be scaled. For example by simulating more deeply the phenomenon by adding more physics, switching to 3D, lowering the time step, augmenting mesh precision... Larger machines are therefore great opportunities as they open the way to greater simulations. How-

[2] NP-complete for example

ever, we shown that unbounded scaling requires a particular effort over parallel overhead: it must remain constant per core in order to allow increasing problem sizes (with p).

For example, from a communication point of view, a given process has to communicate with a constant number of neighbours — forbidding collective communications. More interestingly, collectives are not strictly forbidden as far as they involve a constant number of processes. From a general point of view there are very few programs satisfying these conditions, mainly because of collective communications (generally MPI_Allreduce) at each time-steps, commonly used to compute the next time-step while satisfying the CFL condition[3]. Consequently, finding numerical schemes completely avoiding collectives might certainly be an important milestone on the road to Exascale.

In this context it is then crucial to capitalise development processes — preparing for change. This work is by nature interdisciplinary and crosses individuals boundaries, requiring a strong coupling between scientists which recursively have to produce the best results in order to produce the best results. More particularly, performance-tools and the work of this thesis are part of this process as they locally guarantee efficiency on a link of the simulation chain. They provide feedback to experimenters, positively influencing programs and by extension the modelling process. Next section will focus on program correctness which is of primary importance as there is no need to scale while producing the wrong result.

4.2 Programs Correctness

Computer programs are well known to be subject to *bugs* which cause them to fail or produce erroneous results. More generally, *bugs* can denote everything which causes a computer not to do what the end-user or programmer wants, for example: program interruption (commonly called a crash), producing the wrong results (algorithmic defect), incompatibility with user inputs (interfacing problem) or inconsistent behaviour such as lack of reproducibility or deadlocks [4].

4.2.1 Overview

The term *bug* is relatively vague and seems to describe something latent and almost unavoidable, just like small bugs lying deep in the code-base, randomly impacting programs execution. In other words, programmers, particularly in parallel, have to implement both the feature and its computing substrate. Process which is similar than linking a set of components altogether in order to *promote* desirable behaviours while *preventing* undesirable ones. In other words, programmers are supposed to express both what they want and what they do not want. This second aspect being the most demanding as it supposes that programmers are able to understand and predict any program state. Guaranteeing that they don't lead to a faulty state. In this context, *debugging* is observing a faulty state in purpose of preventing it by fixing a *defect*. In the rest of this section we will rely on the terminology proposed by Zeller [Zel09]:

[3] Courant-Friedrichs-Lewy condition which guarantees numerical problem convergence.

[4] A program deadlocks when it stays blocked in a circular waiting state which cause an infinite waiting preventing the program to terminate.

- **Defect:** involuntary or unforeseen code statement, combined with a given context causes an error in program state.
- **Infection:** the faulty-state is propagated through function calls and parallel interactions, leading to an undefined behaviour. Note that sometimes, the infection phase can hide bugs (for example by rewriting the erroneous value), not necessarily leading to a failure.
- **Failure:** eventually, the defect becomes visible to the user as a failure, associated with an erroneous outputs or crash — finding their root in a defect.

Consequently, a failure can find its origin in a defect, apparently not semantically linked, because of complicated infection mechanisms as, for example, stack or memory corruption. This propagation is potentially both temporal and spatial. In such case, different code sections are being successively called and messages (with erroneous data for example) are sent to remote processes. Therefore, back-tracing the causality chain which led to an erroneous state can be costly and complicated — requiring specifically tailored tools.

One difficulty is that the programmer does not know what he is looking for, consequently, debugging generally starts from the faulty state either reported by users or encountered by the programmer. Requiring to be able to *reproduce and isolate the failure* in a compact test case. Then, once the error is *reproducible*, the developer lists *possible infection vectors* which could have led to such a state for example by examining function stack or variable values. He proceeds by successively *testing propagation hypothesis* from the most probable (according to his experience) to the less probable using various approaches such as breakpointing, watching variables or producing debug outputs. Eventually, once the infection chain has been identified, the developer *isolates and fixes the defect* which led to the failure. This process is described in a more compact fashion by Zeller [Zel09] as seven steps which initial letters mnemonically form the word `TRAFFIC`:

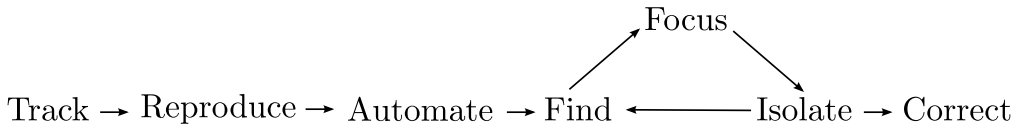


Figure 4.4: *`TRAFFIC` bug-tracking technique as described in [Zel09].*

The process presented in Figure 4.4, and particularly the search loop is very time consuming for developers, Hailpern and Santhanam reported that debugging, verification and testing took from 50% to 75% of the time in typical commercial development organisations [HS02]. Optimising these costs can therefore increase developers' productivity, making compulsory for programmers to be able to efficiency use debuggers in order to speed-up the debugging process.

4.2.2 Quality Process

More importantly, debugging has to be relocated in the *development quality process* which role is to collect, find, fix and prevent program defects at both component and functionality level. Program quality has to be a management-metric in order to incite programmers to

enhance their testing and bug-tracking infrastructure. By systematically relying on reporting front-ends to enforce communication and failure reporting through a *bugtracker*[5]. And making sure software components are not solely tested by their developers but also by users, providing actual test coverage in accordance with the expected functionality. Relying on a modular design with a team for each component can be a possibility to enforce such cross validation. However, it shall be done in a systematic fashion until the integration layers in order to prevent responsibility dilution from "integrators" to "implementors". In other words, the integrated product itself has to be tested by a distinct team. In a simulation context, for example, numericians are in charge of producing results for physicist who can judge their quality and cannot be judged only in the light of incomplete (but necessary) metrics such as convergence. Information management in the quality process shall therefore aim at creating an interdependence between components. Allowing their cross validation and clearly identifying each developer role, globally at component level, and locally at source code level (through versioning and sub-components), making developers responsible for their code and therefore caring for its quality. However, this process has to avoid the stratification danger when a subset of developers is considered as furnishing the core functions and feel free to impose its requirement to the whole project which inevitably looses in orthogonality. Preventing this pitfall requires an ability to preserve individual values in respect to distinct competences while promoting their cohabitation in projects which have to take advantage of individual knowledge and values as rationally as possible, statement which is close to the definition of *management*. Therefore and ideally, code structure shall be mixed with the effective organisation in order to allow component adaptability and preservation of individual values through a management trade-off.

Once collected by programmers thanks to reporting tools and an inciting organisation, defects have to be fixed using classical approaches that we previously summed up in Figure 4.4. This requires, efficient tools to explore program states such as parallel debuggers. Once identified, the defect is naturally fixed and measures are taken to prevent its recurrence by both informing developers (documentation, informal discussion, ...) and adding the issue to the test-base. Then users are informed of issue's resolution. This process aims at preventing software entropy by not putting with a single issue, even if apparently small as it creates a "sense of abandonment" and opens the way for larger issues[6]. This punctual code support is combined with continuous testing where developers rely on several techniques to guarantee programs reliability (see Hunt and Thomas [HT99] which introduces good programming habits) such as:

- **Component-based design:** identify and design in separated components which are connected with compact and clear interfaces.
- **Unit testing:** systematically test individual components in order to make sure they provide and keep providing the expected function alongside the development cycle.
- **Design by contract:** as introduced in the Eiffel language [Mey97](p. 331), document and test every preconditions, post-conditions and invariants in order to make code more reliable by *crashing early* [HT99](p. 114) with a clear error which prevents state infections.

[5] Ticket based web front-end where users report a program failure they encountered with as much detail as possible in the expectation of having developers' feedback on the issue (explanation, bugfix, advice, ...).

[6] This effect is commonly called the *broken window theory* [WK82, HT99].

- **Regression tests:** making sure programs evolution does not impact features or (re)creates bugs. Therefore, it can be interesting to run a complete regression base against each revision in order to point-out possibly harmful modifications or side-effect which could otherwise make their way to the end-user.
- **Integration tests:** rely on representative use cases to test software modules before releasing them to the end-user.

Far from being exhaustive, this list emphasises that there are methods to guarantee reliability and that developers can always strengthen their quality concerns in purpose of producing better software which is more likely to alleviate larger supercomputers requirements.

We have seen that the debugging process takes an important place in the development process as it transforms a faulty program into a working one which satisfies the requirements. However, and particularly with software products, working is not being reliable. Being subject to parallel execution noise, data-set and user input variations, reliability has to be enforced through careful testing at every component level, in fact just as what is done industrially when building a plane or a car.

4.3 Summary

After introducing the developer tools role in terms of software performance and reliability. This section, developed the notion of quality process which is classically linked to guaranteeing means of control and measure relatively to programs. This process allows developers to *constantly* question important metrics about their program. Allowing an iterative development process which compensates uncertainties by information (for example unit and integration testing, bug tracking, documentation, ...). In this complex process, this thesis proposes means of measure in terms of performance and reliability. Once motivated by control, a transition necessarily need measurement to face uncertainties linked to the complex combination of several values. Therefore, in complement of careful testing of every aspects and constant reporting (both part of the classical development process and not covered in this thesis), it is important to question reliability and performance of simulation programs to face supercomputers evolution.

Part II

Key Concepts and Related Work

CHAPTER 5

Architecture of Developer Tools

After the global introduction of part I, this chapter sets up the specific context of this thesis: developer tools development, with a particular interest for performance and debugging tools. After presenting the canonical developer tools architecture, we detail various instrumentation and data coupling methods commonly used in existing tools before introducing analysis methods.

5.1 Canonical Architecture

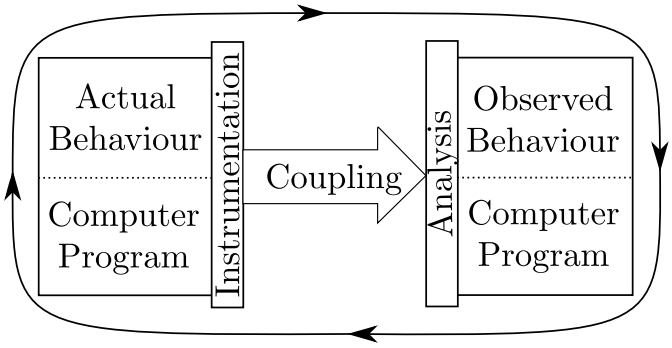


Figure 5.1: *Canonical architecture of developer tools.*

Developer tools can be described in a very synthetic way as a mean which allows an user to observe or interact with a computer program. As presented in Figure 5.1, and highlighted in chapter 4, an effective program behaviour cannot be derived from its code. A tool is therefore in charge of observing this behaviour (measurement), managing both instrumentation data (coupling) and processing those data (reduction, projection) in order to make them intelligible. This process can be seen as a cycle allowing observation and correction of defects through iterative code enhancements. Three classical processes can be derived from Figure 5.1:

- **Instrumentation** captures program's behaviour in order to make it analysable. This process either associates each program step with a dedicated event or makes the program state visible from an external point of view. Process which verbosity defines the overall instrumentation chain requirements (for example in terms of coupling).

- **Coupling** allows the transfer of measurements to a third party tool in purpose of being analysed. This process can take several forms in function of both data volume and interaction/integration levels. Moreover, every tool require a communication channel in purpose of carrying measurements, component which as we will develop, has an important impact over scalability.

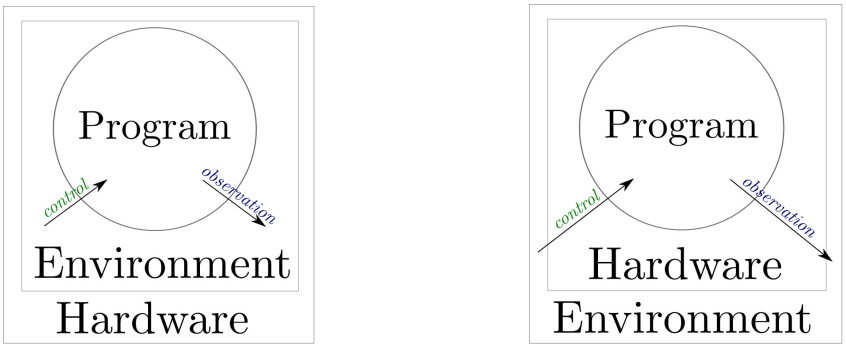
- **Analysis** is the corner-stone of any tool. It aims at projecting observed behaviours on known metrics generally code-related. This process often involves several spatial (over cores or processes) and temporal reduction in purpose of eventually presenting information in an intelligible manner. Consequently, the volume of data presented to users is negligible when compared to the total amount of collected data.

In the rest of this chapter we present developer tools in the light of those three aspects, in purpose of comparing the design choices they incur. We will begin with various instrumentation approaches and then focus on data management methods. Eventually, we detail various analysis principles.

5.2 Instrumentation Approaches

This section presents the wide variety of instrumentation approaches and the spectrum of measurement they provide. In this purpose we outline two classes of methods qualified of external and embedded. (1) *Externals methods* suppose a control over the execution substrate in order to isolate programs in term of state and resource usage. Such approach provides detailed measurements but requires a cooperating execution substrate which in some cases differs drastically from the nominal one. (2) *Embedded methods* rely on an internal program view during its execution, it has the advantage of describing the real substrate but poses the question of measurement perturbation.

5.2.1 External Instrumentation

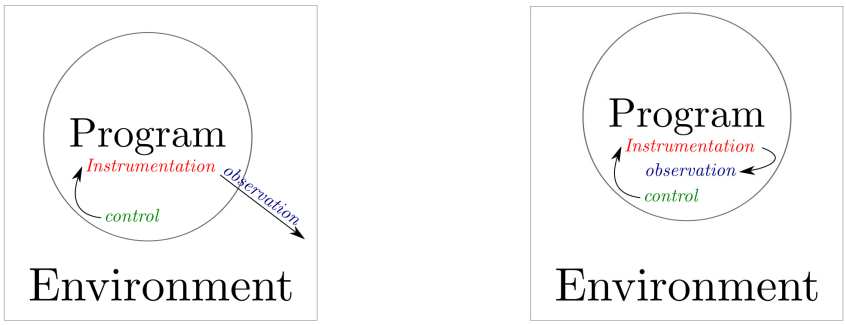


(a) Software Instrumentation. (b) Hardware Instrumentation.

Figure 5.2: *Overview of external instrumentation.*

As presented in Figure 5.2, external instrumentation supposes that the tool runs in an environment surrounding the target application, allowing its separate operation without interfering with the application. This method provides a lot of advantages, among which are both the ability to stop program execution while allowing its inspection and the full cooperation of the executing substrate with the observer. Most of the time, as in Figure 5.2(a), external instrumentation is achieved through execution virtualisation where a software component (instead of directly the operating system) executes a program. Another external instrumentation approach is when the operating system instruments the applications it hosts, combining virtualisation advantages with sometimes close to native execution performance. On hardware side, as presented in Figure 5.2(b), instrumentation can be seen as surrounding the hardware, for example, when using the Joint Test Action Group (JTAG) port of an electronic programmable device for control and inspection purposes. More closely to current processors, hardware counters are also a form of external instrumentation as they provide hardware level metrics which can be retrieved by running applications.

5.2.2 Embedded Instrumentation



(a) Embedded Instrumentation. (b) Embedded Self-Instrumentation.

Figure 5.3: *Overview of embedded instrumentation.*

Figure 5.3 presents the setup used when an application instruments itself. In this case the application is in charge of redirecting its control flow into instrumentation components in order to generate an observable (usually as events). Those events can be either directed in a trace for post-mortem processing (Figure 5.3(a)) or accumulated locally for profiling purposes (Figure 5.3(b)). When using this method measurement-related perturbation might become problematic as there is no way of guaranteeing that the observed behaviour is accurate. Indeed, for example, when storing performance events in a trace, buffer flushes costs are impeded to the instrumented application, delaying local event execution. Difference which as in the butterfly effects can be propagated to every processes through interactions (locks, messages, collectives). Nonetheless, one of the main advantage of embedded instrumentation is that it depicts (even if slightly modified) an actual behaviour as observed on the execution substrate, allowing observations which might not be practical by virtualisation.

The approach consisting in storing measurements (as a trace) for latter processing is referred to as a *post-mortem approach*. It has the advantage of allowing analysis of arbitrary complexity as they are decoupled from instrumentation. Moreover, post-mortem analysis also

opens post-processing opportunities (for example for time-stamp synchronisation) and iterative or comparative exploration. More generally, post-mortem instrumentation stores "unreduced" or slightly reduced events, thus, decoupling instrumentation from analysis and leaving freedom degrees around a common denominator: the trace format. Consequently, trace-based methods can be considered in some aspects as one of the most versatile instrumentation method. However, as we will see in following sections, this approach has drawbacks as, for example, it supposes the ability to manage verbose data in very large traces — causing several data management problems. In particular, post-mortem approaches are subject to the perturbation versus event verbosity trade-off which eventually constrains the set of affordable analysis. Creating a strong need for either online filtering (spatial (per process), per event, temporal (usually trace buffers or sampling), ...) or reductions maintaining a cumulative (profile) state at runtime.

Dealing with local event processing, as presented in Figure 5.3(b), it has the advantage of immediately reducing events by performing analysis "in place", thus, relaxing data management problems. However, such tools cannot perform complex analysis a they directly have to process events with, in general, a limited intermediate storage space. As such tool target runtime instrumentation the retrieval of a global state can be problematic as is goes against the scalability requirement — requiring particular efforts such as suitable topologies (for example tree based overlay networks (TBONS)). Because of those limitations, online approach is privileged by tools which perform lightweight analysis and which are not strongly dependent from a global state such as profilers and validation tools. Nonetheless, some validations such as deadlock detection, requiring a global state, have been successfully developed either centrally or through a TBON (see section 6.4.4).

5.3 Coupling Methods

As aforementioned, the instrumentation process generates a large amount of information which has to be processed in purpose of generating valuable metrics. This section analyses mediums and processes which allow instrumentation–analysis coupling. We detail the three main approaches adopted by tools: (1) analysing data in place, (2) storage of event traces for latter processing or (3) online processing of data (with dedicated resources). Methods which are not mutually exclusive.

5.3.1 In-Place

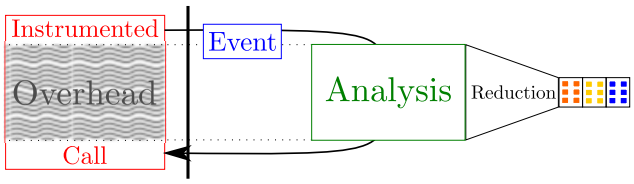


Figure 5.4: *Schematic representation of instrumentation data processed locally.*

Figure 5.4 schematically represents the data-flow and control flow associated with local instrumentation data processing. As described in previous section, analysis overhead is directly

impeded to the instrumentation wrapper as the control flow is rerouted to analysis routines. Dealing with the data management aspect, instrumentation events are immediately projected to compact metrics (for example, a profile, arguments validation, ...). They are stored on the stack and directly passed to analysis routines which immediately perform their projection on compact metrics (either spatially, temporally or functionally, see section 5.4). In summary, in-place analysis is the most space efficient approach as data are not stored at all, however, projection costs have to be limited, restraining analysis verbosity. Moreover, by nature such approach only allow the generation of either punctual events (issuing warnings, aborting, ...) or reduced events (cumulative state must fit in programs memory) — preventing for example exploratory analysis.

5.3.2 Post-Mortem

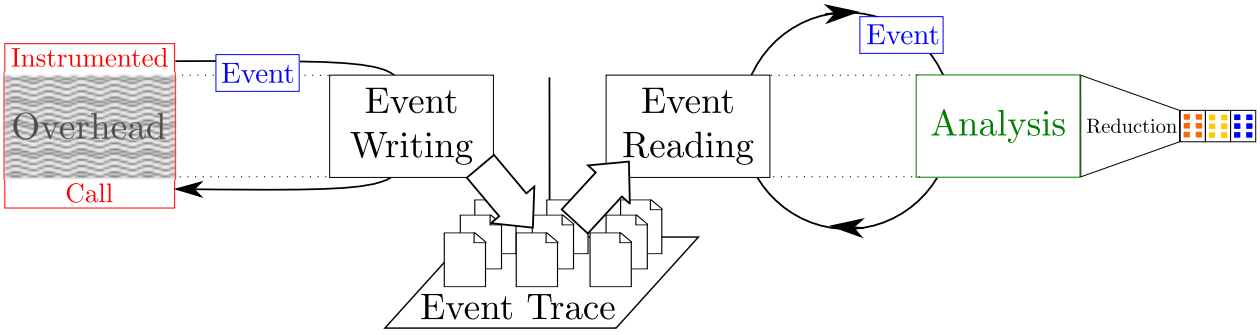


Figure 5.5: *Schematic representation of instrumentation data post-mortem processing.*

As presented in Figure 5.5, trace based approach consists in storing events outside of the parallel programs (generally in a file based trace) in order to process them separately. This process, allows complex analysis as they wont disrupt the execution. It also open possibilities for successive analysis on the same data-set. However, managing large traces can be quite challenging as they grow rapidly with both event verbosity and the number of core[1]. This poses the question of management and post-processing of those large data-set which are challenging payloads for peta-scale file-systems. Consequently, trace-based analysis requires a specific handling of file-system resources (for example through parallel I/O[2] libraries, see Section 6.4.1) to scale up to a full machines. Moreover, trace processing must be parallel in order to process measurements in a time frame compatible with iterative use.

5.3.3 On-line

The on-line approach presented in Figure 5.6 can be seen as a combination of in-place and post-mortem approaches. Instead of being processed locally, data are sent (for example, through the network or a shared memory segment) to an on-line analyser which is able to reduce data without impacting the application. As the analyser has a limited amount of memory, data have to be either reduced quickly or limited in size in order not to block the application. Moreover, as events are reduced/filtered before being stored, this approach does not allow

[1] Traces of hundreds of GigaBytes are common.
[2] I/O: Inputs/Outputs

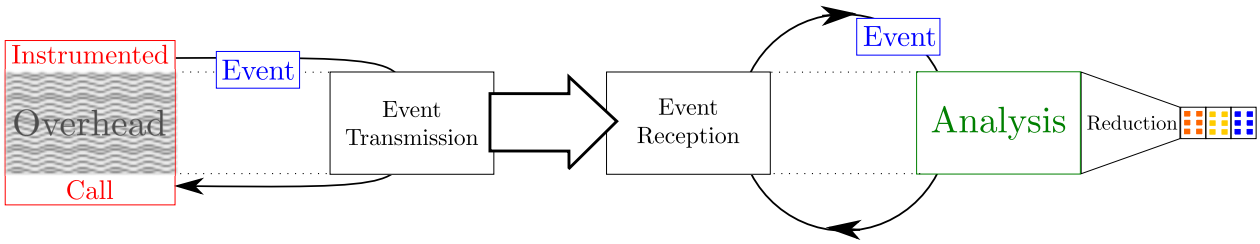


Figure 5.6: *Schematic representation of instrumentation data on-line processing.*

iterative analysis over fine-grained events — preventing exploratory analysis which is always possible with traces. However, due to its on-line nature, this methods does not stress the file-system as in the trace based approach and allows the scaling of analysis resources in accordance with both analysis costs and data verbosity. Indeed, the IO budget is determined by the number of servers providing this service, value which is generally a limited portion of the machine. Whereas, the on-line approach takes its resources (including bandwidth) from the computing partition with the possibility of reaching higher performance.

5.4 Performance Event Analysis

Performance analysis consists in *reducing* events to synthetic metrics which can be easily matched with possible program improvements. It projects a large volume of individual events over various models which capture important execution aspects while remaining understandable by users. If we consider the parallel execution as a whole it can be seen as a multidimensional experiment which can be projected on a wide range of metrics:

- **Space:** distributed programs are running on a discrete set of machines. Moreover, as we previously described, supercomputers have both intra-node (NUMA, NUOIOA) and inter-node (network) topologies, definitively "shaping" the execution space. Therefore, *spatial reduction* can provide some insight on computation and data scattering, emphasising phenomenons such as imbalances or dependency propagation.

- **Time:** by nature, a program is designed to perform a set of operation in a defined order, in the purpose of fulfilling its purpose. Consequently, it is interesting to observe computation progress over time as it depicts the ability of the program to actually perform its task in terms of both parallelism and synchronisation.

- **Code:** a parallel program execution is always preceded by its expression as an human readable code. Therefore, projecting the execution on program sources can be an efficient way of providing feedback to programmers. Expressing measurements in an actionable language (iterative code modifications, coupling with version control to identify regressions, hotspots identification, ...).

- **Programming models:** parallel execution is strongly coupled with the way parallelism is expressed at source code level. As there are several parallel programming models and that their mixing becomes the norm (see Section 2.3) it can be useful to correlate programs behaviour with individual models. Not only for validation purpose but also at performance level, for example by identifying missuses of MPI, OpenMP fork-joins, locking schemes...

An important aspect of performance event analysis is that it aims at providing the user with intelligible metrics, *summarising* the parallel execution. Consequently, analysis is generally a destructive operation (except in some rare cases of trace visualisation with tools such as Vampir) which operates a reduction of individual events over the aforementioned metrics. In this context, when comparing profiling and tracing, analysing performance data is a way of reducing their volume through what can be described as a computation–performance trade-off. Consequently, where the analysis takes place is crucial to budget important factors such as overhead, analysis complexity and performance data verbosity. For example, reducing data early is a way of avoiding unneeded data transport while producing valuable measurements. However, as this reduction is a destructive process, it prevents further analysis and exploratory approach. Eventually, another consequence of early reduction is that every analysis [3] must have its associated storage description in order to be transferable to the user either as a report or through a graphical interface. This makes tools performing early reduction [4] quite dependent from their underling performance description scheme, possibly complicating analysis extension as it can impact the whole measurement chain.

5.5 Summary

This chapter presented the context of developer tools, and particularly performance and debugging tools from a global point of view. We introduced the two main instrumentation approaches which either consist in instrumenting either internally or externally. Then we detailed the three different coupling methods, processing data locally, through post-mortem traces or remotely via an on-line coupling method. Eventually, we described the analysis process as a reduction which *projects* performance data over intelligible metrics in order to be fully understandable by the user — process which can also be an opportunity for performance gains. Consequently, this chapter underlined the fact that there are several design alternatives to implement performance tools, offering tools developers a wide range of alternatives.

[3] Apart the trivial case where a profile is displayed on STDOUT.
[4] Mentioned later as hybrid Instrumentation Chains in section 6.1.2.

CHAPTER 6

Related Work

This section presents both existing and related work for different parts of our contribution. We start by introducing existing developer tools with debuggers, and then pursue with profiling and validation tools. Then, we detail complementary subjects which are time-stamps synchronisation, blackboard systems and data-management approaches. This chapter gathers most of the references when dealing with the state of the art itself, several references of corollary subjects can be found in other parts of this manuscript. Note that back-links are available in the bibliography in order to explore citation sites.

6.1 Developer Tools

This section presents the main classes of developer tools, insisting on their functions and common implementations in term of instrumentation, coupling and analysis. We successively introduce debuggers, performance tools and validations tools. In each case we recall some context, expectations and advantages associated with existing approaches.

6.1.1 Debuggers

Debugger are the most commonly used developer-tool as they are an efficient way of diagnosing a faulty program in order to reestablish a feature which is the reason to be of a computer program. Therefore, programmers generally start from a faulty state and re-launch their program using a debugger in purpose of exploring its state. Consequently, debuggers must be able to describe faulty program state, to do so, they generally rely on the ptrace [HC99] system call which allows a parent process to observe and control one of its child. This single system call provides enough fuctionalities to build a full featured debugger such as GDB [Pro13b], IDB [Int12a], LLDB [Pro13c] or DBX [Lin90] for example: acessing child's memory or registers, retrieving signal informations and controling child's execution (trough signals). Other approaches can also rely either on sampling to continuously collect call-stacks as in the STAT debugger [AdSL+09] or, more commonly, use crash-dumps to collect programs final state. More particularly, some debuggers can take advantage of the JTAG [LG98, Gra01] or serial port [Axe07] of electronic devices to perform debugging from an external point of view. Debuggers cover a wide range of the instrumentation spectrum, with *on-line approach* generally via ptrace [Pro13b, Int12a] or sampling [AADS+07, AdSL+09] followed by a reduction through a TBON [RAM03], *post-mortem* approach with crash dumps or traces [Rei93, Tea06, HM01] or even *in-place* debugging using back-trace libraries such as libunwind [lt11] or libc's backtrace function [KP07] from a signal handler.

Dealing with the interfacing, debuggers generally provide a command line interface [Pro13b, Int12a, Pro13c, Lin90] for common use at single node scale. However, when dealing with parallel payloads, displaying programs states becomes a challenging task which requires a scalable design not only to collect data but also to display them. Tools such as net-DBX [NE01], p2d2 [Hoo96] or Panorama [MB93] debugger relied on a client server approach where a graphical user interface could be attached to several processes. However, the increasing number of cores made the use of scalable communication topologies compulsory. Therefore, Tree Based Overlay Networks (TBONs) such as Paradyn MrNEt [APM06, RAM03, Rot05] became a common intermediate in scalable debuggers for both control and reduction purposes. Such approach proven its scalability with the STAT debugger [AdSL+09] which relies on MrNet [RAM03] as it has reached the petaflopic scale on the Sequoia supercomputer. Dealing with commercial MPI-aware debugger such as Allinea DDT [All13a] or Totalview [Sof13] which are commonly used on petaflopic range supercomputers, they also rely on a tree based topologies, reducing data before displaying them in a graphical interface.

6.1.2 Performance Tools

Performance tools can be sorted in two main categories, on one hand those which rely on fine-grained events through tracing and on the other hand those which process data locally. However we will see that some tools, combine these two approaches in a third category, in purpose of getting the best of both world, for example, by reducing an event subset and tracing another.

Trace-Based Approach

The trace-based approach which consists in storing events in a file-based trace for post-mortem processing requires a *trace format* which is in charge of defining an efficient storage layout in term of storage and scalability. Consequently, a trace format shall not only contain time-stamped events but also meta-data describing the execution context (topology, symbols, ...). From a more general point of view a trace format have to satisfy the following aspects:

- **Consistency:** a trace have to be readable out of its original context. It shall be processes later after the execution without information loss. This supposes that the trace format embeds specific contextual informations (symbols, topology, timing, ...).
- **Scalability:** as tracing targets are highly parallel programs. Analysis has to be scalable at both writing and reading time. Supposing particular efforts on IO management, preserving an end to end parallelism.
- **Compact data:** important data size generated by tracing tools is of the main limitation for trace-based approaches (data management and its associated overhead). This makes efforts to restrain instrumentation data size compulsory.
- **Topology handling:** tracing tools generally define execution stream topology as an acyclic graph relying on $(\mathtt{parent}, \mathtt{child})$ relationships stored within the trace.
- **Events:** the core of a trace format is the set of events it supports. Formats generally define event semantic by providing common events with a fixed definition (for example, MPI interface). It shall be noted that this specialisation of trace formats poses questions of format adaptability.

- **File Handling:** as a trace format aims at storing data in a parallel file system, particular efforts are required to guarantee a minimum of scalability, for example, by limiting the total number of open file descriptors.

In addition to those common concerns, the way analysis are processing instrumentation data necessarily constraints their layout. Indeed displaying a temporal trace with the ability to scroll or zoom in and out makes temporal look-up crucial (as in the SLOG2 [CGL08]). Whereas, such format used for debugging might suffer from limitations, for example, when processing a single type of event which being mixed with others might require a complete trace walk. There are several trace format (one for each tools ?), however, some formats became more widespread than others, setting a common ground for performance analysis and as we will see opening interesting interfacing opportunities between tools.

Trace-Based Tools

An extensible trace format which relies on a meta-description approach in the purpose of instrumenting multi-threaded programs is the Pajé [dOSdKM10] trace format which is associated with a trace-visualisation tool [DKdOS00] for interactive exploration of event traces. Despite its versatility, the main default of the Pajé trace format is its text based (ASCII based) storage approach which privileged modularity over space efficiency. Consequently, Pajé is with no doubt one of the most extensible trace format but it lacks of a proper compression and parallel IO infrastructure to be suitable for massively parallel application tracing. Moreover, parsing a text-based trace format costs generally more than reading a binary format which can be immediately matched with a C structure.

Another trace format which has been used to trace parallel applications is the Open Trace Format (OTF) [KBB+06, KBMS06]. This format relies on an ASCII-based storage method which stores values without leading zeroes (for compression purposes). Writing and reading are performed using a state machine which describes event layout. At the beginning of this thesis, OTF was the state of the art trace format, as it was used by production grade tools such as Vampir Trace [KBD+08] to store massively parallel event traces. OTF provides both reading and writing primitives to handle individual events stream and supports compression using zlib [DG96]. It provides predefined events for common MPI calls, efficiently instrumenting parallel applications while supporting fast event look-up in terms of both time and space (between processes). However, OTF does not handle parallel IO libraries, preventing its use at larger scale as the large number of files (one per process thus one per core) can saturate file-system meta-data servers. Another limitation is the identifier handling which upon trace collection are all local to their stream, requiring a full trace rewrite to *unify* collected data. OTF2 [EWG+11] which succeeded [1] to OTF brought several improvements. It added the support for the SIONlib [FWP09] which provides parallel IOs and therefore leverages the problems of scalability associated with the number of files. Moreover, OTF2 storage format moved to binary, evolution which drastically reduced traces storage size (see section 9.4.6). Dealing with the identifier unification "problem", OTF2 features a direct conversion approach through a local to global mapping table which avoids file copies.

[1] OTF2 was not available at the beginning of this work. Otherwise, as discussed in the limitation section of our trace based approach we would probably have used it to propel our tracing tool.

SLOG(Scalable LOG file) [CGL08] is a trace format specifically designed to handle temporal traces visualisation. Events are stored in the form of a binary tree which defines temporal intervals in the visualisation window. Therefore, zooming in and out is choosing a node in this tree, node defining a bounding box matching current viewpoint. Approach used in complement of SLOG in the Jumpshot [ZLG+99, WBS+00] trace visualisation tool.

The EPILOG (Event Processing, Investigating, and Logging) [WM04] trace format has been developed for the KOJAK [MW03] measurement infrastructure, it relies on a binary data format and supports both MPI and OpenMP hybrid codes including performance counters,thanks to the PAPI [MBDH99] library. The KOJAK performance tool set (which is the precursor of Scalasca) allows OpenMP instrumentation programs using the Opari [MMSW02] source translation tool-chain in order to insert instrumentation calls. It stores events within traces which are processed using EARL [WB04] which is the high-level interface for accessing EPILOG traces. It defines event abstractions (a hierarchy of event types) and provides program state handling (stacks, messages) with random access capabilities. Thanks to this high level interface, the EXPERT [WM03] analysis tool is able to process the trace in order to generate a compact representation of performance information which can be visualised using CUBE(CUBE Uniform Behavioral Encoding) [SW04] visualisation tool. CUBE relies on three panes to present performance data in a compact manner (1) metrics, (2) call-tree and (2) location, allowing the exploration of measurement data in the CUBE performance space [WM03, Wol03].

Trace visualisation tools such as Vampir [KBD+08] trace are able to visualise OTF traces, collected either with the libVT (included in OpenMPI), producing OTF1 traces or more recently with the ScoreP measurement system (see next section) which produces OTF2 traces. Vampir allows interactive exploration of large event traces which can be augmented with several performance metrics (hardware counters, communication matrices, ...). To face the challenge of displaying very large traces, the Vampir GUI can be used as a client to the VampirNG [BM08] trace analysis engine which performs a parallel trace processing of OTF traces. Vampir also has its own trace format called VTF3 [SKMP04]. A non-commercial alternative to Vampir is Vite [CDFT12] which also allows interactive visualisation of OTF traces.

Paraver [PLCG95] is a performance visualisation tool which relies on both its own trace format and instrumentation layer called EXTRAE [BDMQO12] which brings support for Pthread, OmpSs, OpenMP and MPI. Paraver has the particularity of allowing the user to build its own performance metrics in purpose of exploring program states using a powerful filter functionality. Paraver is also able to process multiple-traces in parallel, feature which can be useful for example to compare two versions of the same code.

Valgrind [NS07b] tool, callgrind [Wei08] can perform profiling of programs thanks to the valgrind infrastructure. Despite a relatively important overhead and a limited support for parallelism (due to virtualisation), callgrind and its associated visualiser Kcachegrind are one of the most comprehensive profiling tool, providing in the same tool, callgraph visualisation, visual profiles and performance metrics projection at source code level.

The paradyn [MHC94] tools are a set of tools which rely on binary instrumentation thanks to the dyninst [RBR+07] tool in order to instrument unmodified executables. Paradyn pro-

vides performance analysis though a parallel "Performance Consultant" engine [MCC+95]. The paradyn team is also at the origin of the MrNET [RAM03,JBM12] TBON framework with propelled the STAT debugger [AdSL+09] at petaflopic scale. If we compare Paradyn which our on-line approach, we can see that it also performs a runtime coupling thanks to the MrNet [RAM03] framework which itself relies on TCP sockets (MrNET 4.0.0). Consequently, as we will further develop in Chapter 10, our method provides support for high performance networks (thanks to the underlying MPI) and is build around a distributed data-flow engine which simplifies analysis specification.

Hybrid Instrumentation Chains

In order to reduce the volume of data stored in performance traces, some tools adopted an hybrid approach by combining both profiling and tracing. Therefore, most performance event are reduced in place, for example by being projected on a call-path profile, whereas, a manageable subset of events is actually stored for latter processing (for example MPI communications). If balanced correctly in terms of embedded analysis cost, this method can provide valuable measurements while remaining scalable and non-invasive.

OTF2 has been included in the ScoreP [aMBB+12] measurement system which gathers several profiling tools (Vampir, Scalasca, Tau, Paraver) around the same measurement infrastructure and trace format, allowing users to use several tools on the same trace file. Opening opportunities for complementary use and interaction between tools. ScoreP features state of the art instrumentation capabilities, including OpenMP tasks [LPSW12] and is able to generate OTF2 traces, Cube profiles (see Scalasca subsection) and Tau profiles (see Tau subsection) — both making of ScoreP the most versatile instrumentation library and filling the gap between tools which were isolated because of their different trace formats[2].

Scalasca [GWW+10] is the successor of KOJAK, it relies on an hybrid approach which combines both profiling and tracing in order to produce valuable performance metrics with a reduced overhead. Functions calls which are among the most verbose events are reduced in place using a profiling approach which can be made even more lightweight thanks to sampling [SGS+11, SWW11]. Dealing with MPI events stored for post-mortem processing are processed in an original fashion as they are replayed upon application end, in purpose of generating performance metrics such as wait-state analysis [GWWM09]. As Scalasca also subject to the problematic of identifiers unification (see OTF trace format), a scalable hierarchical approach [GSS+12] had to be developed in order to scale to larger systems [WGM+10]. Dealing with analysis, Scalasca relies on the CUBE [SW04] visualiser, enriched with several performance metrics such as Wait-states [BGWA10] analysis, load-imbalance [BWG12], one-sided communications support (thanks to the replay approach) [HKW11], performance dynamics [Sze12] (to our knowledge not included yet in current release 1.4.3).

Dealing with the TAU [MMSH10] performance tool-set, it is probably one of the most versatile one. It supports a wide range of instrumentation methods among which are directive rewriting [MMSW01], function and loop instrumentation [JDA+09], GPU support [MBS+11] and sampling [SMH98]. On the analysis side TAU relies on the PerfDMF [HMBM05] performance data management framework which provide TAU analysis with a common storage

[2] Trace format converters are available, but who would convert a 100GB+ trace ?

and data access infrastructure. TAU supports snapshots [MSMS08] which can be viewed as sampled profiles, it also supports phase based profiling [MSM05]. The Paraprof tool can be used to explore performance measurements by displaying profiles, time matrices, call-graphs and also features an interactive 3D visualisation tool [SML+12]. TAU also relies on PerfExplorer [HM05] for original performance analysis capabilities [HMSM07], including data-mining in-between application runs. TAU provides support for run-time monitoring [SMS99], for example with the MrNET [RAM03] TBON framework [NMM+08, LMM11].

Static Analysis

Static analysis consists in deriving information from the binary without requiring program execution. This approach has the obvious advantage of relaxing hardware dependencies, allowing program projection in any context as far as it has been modelled. Moreover, this method forces the setup of a symbolic execution model which is an important field of research as machines are becoming more and more complex. We cannot solely rely on empirical metrics (profiling) and it is important to define what can be expected from a given platform to initiate a proactive approach toward the computing substrate — encouraging the advent of new programming models while helping during the machine/hardware design process. However, static analysis can be very challenging, not only because it relies on low level representations (mostly source code or sometimes the binary itself) but also because it faces execution combinatorial aspect which arises from both parallelisms and architectures complexity (caches, prefetching, branch prediction, ...).

MAQAO [DBC+05] is a tool aimed at optimising binary code. It relies on a powerful static analyser to disassemble the binary in order to rebuild the control flow graph (CFG). Dissassembler which provides an instrumentation interface and has OpenMP support [BRJ+10]. On top of this infrastructure, MAQAO provides a plugin framework in order to support different types of analysis [SCOJ13]: a static architecture performance model, STAN for performance tuning hints, DECAN [KZO+10] which allows decremental analysis and memory based value profiling. More generally, compilers such as GCC or ICC are also relying on performance prediction, for example to choose between optimisations alternatives (such as unrolling factors), choices made from heuristics which are derived from static analysis.

On-line and In-place Tools

As detailed in previous chapter, on-line approach handles performance data using computing resources distinct from those on which the program runs, thanks to a coupling mechanism which forwards data to the analyser (generally through the network or via shared memory segments). This approach avoids file-system bottleneck while preserving fine grained events. It also opens opportunities for “real-time” profiling (introspection). Dealing with the profiling approach, it consists in reducing events locally by directly projecting on performance metrics (profile, counters, call-path, ...), thus, completely avoiding performance data manipulation. However, as profiling cost directly impact instrumented application, analysis have to remain simple (in terms of computational complexity) and independent from a global state.

Periscope [BPG10] is a tool which performs online automatic analysis of parallel programs. It relies on a tree of agents performing a reduction on performance metrics, displaying them in an interface fully integrated in the Eclipse integrated development environment. It can

perform (among others) memory accesses analysis [GK07] or finds inefficiencies in the use of OpenMP [HSC+08]. Analysis modules can rely on a powerful performance specification language ASL (APART Specification Language) [GF07] which allows automated performance analysis [GFK05, FG07].

mpiP [VM01] is an MPI profiling tool which relies on a statistical aggregate of communication operations. Profiling data are collected locally to each task and reduced at the end of the execution. Thanks to its lightweight approach mpiP has less overhead while providing resuls close to the effective execution. HPCtoolkit [ABF+10] relies on sampling combined with stack unwiding and performance counter collection to generate scalable MPI program profiles, with support for node level parallelism [TMC09]. For example, it can generate call-path profiles [AMCT10], analyse lock contention [TMCP10] and load imbalances [TAMC10].

6.1.3 Validation Tools

Validation tools aim at projecting programs behaviours on their underlying model, therefore, producing highly valuable errors which can be immediately matched with actionable concepts. At the difference of profiling which only displays the consequences of a defect, validation's purpose is to find the cause of this defect — directly yielding valuable information. Consequently, there could be as many validation tools than programming model or concepts, here we present those we are directly concerned with: parallel programming models (MPI, OpenMP, Pthread) and memory management.

Message Passing Interface

Marmot [KMR04a] is an MPI checking tool which validates the use of the MPI interface using the PMPI interface. It allocates an extra process which takes care of analysis such as deadlock detection which require a global state. It support hybrid OpenMP programs [HMK09] and provides feedback on several types of events such as MPI IO [KMR04b] or one sided communications [KR06]. Marmot is able to generate comprehensive HTML reports and have been included in both the DDT debugger [KHL+07] and the CUBE visualisation tool. Umpire [VdS00] is quite similar to Marmot as it provides MPI checking capabilities, however, unlike Marmot, it is limited to shared memory platforms as tasks communicate through a shared-memory segment, the manager being a thread in task 0. Umpire is able to perform several MPI related checks, including deadlock and mismatched collectives detection. Similarly MPI-CHECK can be used to validate the MPI Fortran 90 interface by building a "knowledge base" of MPI calls which is used to instruments individual MPI call at source code level. Other tools such as the Intel Message Checker [DKDS+05](IMC) can provide such analysis at MPI layer level. IMC has also been included in the DDT parallel debugger. Eventually, Some MPI implementations also feature checking methods such as MPICH for MPI program correctness [PGK+07].

Another tool which succeeded to Marmot and Umpire is MUST which provides features similar to those of marmot but with an extended scalability thanks to a tree based overlay network (or TBON) architecture. The Generic Tool Infrastructure (GTI) [HMdS+12] which relies on P^NMPI and provides a generic infrastructure for instrumentation and event reduction purposes allows efficient generation of event instrumentation, transport and reduction through XML specifications, allowing the offload and parallelization of MUST validations. The GTI is build over P^NMPI [SdS07] which is a framework allowing the stacking of several tools at the

PMPI interface level. Moreover, P^{N}MPI introduced an MPI virtualisation approach which consisted in wrapping the whole MPI interface while replacing references to `MPI_COMM_WORLD`. Idea which motivated our on-line tracing approach, eventually leading to our stream implementation for on-line profiling purpose (see section 10.2.3).

Thread Level

Some tools also target thread level parallelism to detect programming model misuses or involuntary error such as race conditions. For example the Valgrind [NS07b] tool Helgrind [MW07], is able to detect race conditions using a lockset algorithm. Other tools such as Thread Sanitizer [SI09] or Helgrind$^+$ [JT08] extend Helgrind's lockset approach with an happened before relationship to reduce the number of false positives. Sun Thread Analyzer [Ora07] or the Intel Thread Checker [BBMP06, PS03] can perform either race condition or deadlock detection on either Pthread and OpenMP programs, both featuring a graphical user interface allowing faulty code exploration.

Memory

As memory errors are very common in computers programs (leaks, double free, unauthorised access leading to a segmentation fault...), several validation tools were developed in order to help programmers to diagnose and fix those errors. For example, the Memcheck [NS07a] tool which is part of the Valgrind [NS07b] framework is able to track most memory related errors at the cost of a relatively important overhead and limited parallelism support due to Valgrind's virtualisation approach — drawback which is also a strength when coming to measurement accuracy as it allows the instrumentation of every load and stores. Another approach used by AddressSanitizer [SBPV12] (ASan) relies on the LLVM compiler to instrument memory accesses in purpose of producing errors similar to those of Memcheck, minus the consequences of virtualisation. Debug allocators such as Electric Fence [Per03] add restricted guard pages before and after each allocated segment in order to catch the exceptions associated with out-of-bounds errors. These tools provide less features than memory validation tools which are able to instrument load and stores. Moreover, they increase program memory consumption (guard pages) and slow down allocation because of the systems calls required to set guard pages permissions (mprotect). Eventually, other debug allocators such as DieHard [BZ06] can rely on canary (or 'magic') values around allocated segments to detect out-of-bounds modifications.

6.2 Time-stamp Synchronisation

Synchronising clocks is critical to allow temporal analyses or rendering which require an accurate global distributed event view. This section describes common clock synchronisation techniques. We first outlines the importance of time-stamp synchronisation for performance analysis. Then we present common time sources before describing time-stamp handling in instrumentation context.

When running a program on a supercomputer and therefore on several computing nodes, synchronising clocks is compulsory to acquire a global temporal view. Although some supercomputer architectures such as the Blue Gene/P have a centralised time source, providing user with an accurate global time [CBC$^+$05], most platform does not offer such facilities. In general, the most precise time source is available at processor level (see next section). It is

synchronised between the cores of a given node but not in-between nodes. In this context, events within the same node can be observed at clock resolution (close to nanosecond), allowing a fine-grained event study. Whereas, there is solely the "happened before" relationship for guaranteeing time-stamps outside of nodes boundaries, for example, through a Logical Clock (LC) [Lam78].

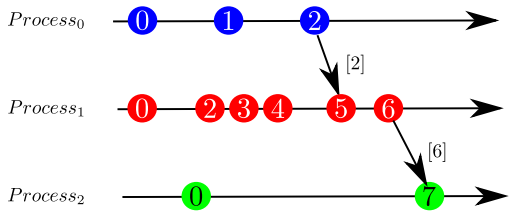


Figure 6.1: *Example of Lamport Logical Clock [Lam78] for tree communicating processes.*

As presented in Figure 6.1, a logical clock ensures causality between two interacting processes but does not ensure causality between processes which do not communicate. This logical clock has been extended to a vector clock [Mat88, Fid88], not only propagating a single clock but the whole "synchronisation vector", allowing a stronger ordering propagation. When targeting profiling applications, this method cannot be satisfactory as it does not provide duration information for inter-nodes events. For example, in two communicating nodes, a logical clock is not sufficient to qualify messages latency or collectives duration, which by nature are distributed events. In such context, a coherent time source with an error lower than observed event duration (messages latency is in the μsec range) is required to provide a coherent view outside of nodes boundaries.

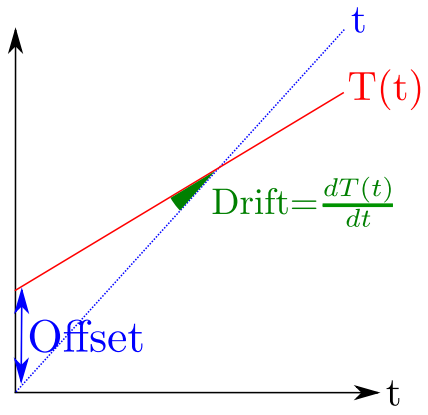


Figure 6.2: *Parametrisation of clock error assuming a linear representation.*

Assuming a linear representation of clock time $T(t) = at + b$ with a the drift factor and t the time offset, the error can be described by these two parameters. As presentedÂăin Figure 6.2, the *offset* is the absolute difference between clocks and the *drift* is their frequency difference. Naturally, this approach is only a model as in practice clocks are not linear, depending on several factors. For example the temperature which influences oscillator frequency, creating non linear errors. However, in a first approximation, clock frequencies can be assumed to be

uniform on a given interval of time. But, provide time references which are bound to diverge after a sufficiently long time. Correcting drift variations errors requires much more computing efforts and complex algorithms such as Scalasca's amortisation [BRW07, BLRW08, Bec10]. The Intel trace analyzer and collector [Int12b](section 5.1) relies on a linear interpolation in-between timestamps taken at the beginning and upon program completion, method also used by Vampir trace which can also extends the process by resynchronising at every global collective [fISZ13].

6.2.1 Time Source

Our instrumentation relies on the most precise time source available in current supercomputers: the TimeStamp Counter (TSC). This counter, available on most processors, runs at a frequency close to the processor one, making it by far the most precise time source available ($f_s = 2.8 \times 10^9 \rightarrow T_s = 0.35ns$) with resolutions in the nanosecond range. It is generally a 64 bits counter, incremented at a constant frequency f_s since the machine has been started. Its value can be retrieved through an assembler call similar to the one of Figure 6.3 or using the *cycle.h* header from the FFTW project which provides a portable implementation. On recent architectures, this source is synchronised in-between cores, providing a reliable time source at node level [INT10].

```
static inline uint64_t get_TSC( void ) {
  unsigned a , d;
  asm("mfence"); //Memory Barrier
  asm("lfence "); //Load barrier
  asm volatile ("rdtsc" : "=a"(a),"=d"(d )) ; //TSC retrieval
  return (a)|(((d)<<32) ;
}
```

Figure 6.3: *TSC Retrieval on the x86-64 architecture.*

6.2.2 Synchronisation

Dealing with the synchronisation process itself, feedback loops based on round-trip estimation, derived from Cristian's synchronisation algorithm [Cri89] are the most widespread approach. Such method is for example used by the Network Time Protocol (NTP) [Mil91] and has been used by previous versions of the Linux kernel [Kle05] (before being removed because of hardware synchronisation [Mol07]). A study, of global time round-trip based synchronisation over given topologies (fully-connected, ring, star and hypercubes) has been done by Jezequel [Jéz89]. Similarly, Dunigan analyses such synchronisation for hypercubes [Dun92]. Several alternative methods are proposed, including the one sided synchronisation approach proposed by Drummond et al. [DB93]. It relies on point to point communications at the condition of being able to derive a communication pattern involving every processes. Method limited by latency variations (no feedback), messages heterogeneity and one-sided communication patterns diameter which, by nature, are decentralised. Dealing with hardware related methods, Liao et Al. [LMC99] achieved a $1\mu s$ accuracy relying on a specific Myrinet network support (*MyriTime* packets) and Cristian's algorithm [Cri89]. Sensors network often rely on network layer broadcasts to perform distributed synchronisation either in single hop [VCR93, VRC97, HC02] or multi-hop manner [EGE02]. More generally, one of the most widespread use of distributed time synchronisation is with no doubt the Global Positioning

System (GPS) [KH06] which relies on tight synchronisation thanks to atomic clocks in order to provide its positioning service.

6.2.3 Logical Clocks

As far as logical clocks are concerned they were introduced by Lamport [Lam78] as simple counters guaranteeing causality in-between interacting processes. By nature, such counter do not offer time measurement capabilities as events are numbered in order of occurrence as monotonous sequences. This first logical clock only propagated interacting process status, neglecting its previous temporal context. This limitation has been addressed by vector clocks simultaneously designed by Mattern [Mat88] and Fidge [Fid88] by propagating the whole set of temporal constraints, yielding tighter causality bounds. Vector clocks are strongly consistent as they accurately capture causality. However, Charron-Bost shown that their size in purpose of capturing causality cannot be less than n [CB91], with n the number of processes, posing the question of their scalability. Logical clocks have the propriety of clearly describing the dependency links in-between events and therefore freedom degrees, propriety which cannot be compactly captured by classical time-stamping, even if arbitrarily precise. Consequently, this approach might be a good candidate for parallel instrumentation, maybe not only as a corrective component but also as the actual timing substrate.

6.2.4 Time-stamps for Instrumentation

Getting back to trace related time-stamp synchronisation, various approach were adopted. A common method consists in synchronising clocks at both program start-up and ending in purpose of performing a linear time approximation, approach used by both VampirTrace [fISZ13] and the Intel Trace Analyzer [Int12b]. Approximation method which statistical effects have benn studied by Maillet et Al. [MT95]. On the opposite, the Scalasca tool-set relies on a replay based strategy which purpose is to reestablish the clock condition for messages through forward and backward amortisation [BRW07, BLRW08, Bec10]. The VampirTrace instrumentation library can also rely on an internal timer synchronisation [DKMN08] which takes advantage of collective operation in order to regularly synchronise time-stamps thanks to a multi-hop algorithm based on a k-regular topology, with the possible drawback of impacting collectives performance. More generally, synchronisation quality is subject to a trade-off relatively to its performance impact. Indeed, when trying to model the impact of a measurement or time-stamp correction, as outlined by Malony et Al. [MR91, MRW92, SM93], several combinatorial aspects have to be taken into account as such correction are not only linked to local parameters. For example when correcting a local clock offset, a special care has to be taken not to violate the clock condition: “a message cannot be received before being sent” — simple predicate which is far from being obvious in presence of non-linear clock effects, for example, because of cascade effects, randomly propagating corrections.

6.3 Blackboard Systems

Blackboard systems are a class of expert system which has been deeply influenced by artificial intelligence (AI) related concepts. Expert systems purpose is to find a way of modelling knowledge in order to help decision in either complex or uncertain environment. Several approaches were developed among which the forward inference model (if–then) used by

both MYCIN [Sho76] which identified bacterias in order to recommend antibiotics and DENDRAL [LBFL80] which made hypothesis on chemical structures. A more recent approach relies on inference engines, formulating hypothesis using either logic or fuzzy logic thanks to specifically tailored symbolic languages such as LISP [Ste90] or PROLOG [Rou75,SSE90]. Expert systems became very popular in the 80's [RN10], a lot of 'shell' were developed in order to build meta-expert systems from higher level values [FG87]. Blackboard systems which found their origins in this context [EL80,EM88], are a kind of problem solving framework where several agents [Cor03] or *Knowledge Sources* (KS) are gathered around a common data-structure or *Blackboard* (BB). Those agents can read and write on this data-structure in order to produce new data which will be at their turn globally available for *opportunistic processing* by other Knowledge Sources. This work-flow, derived from the analogy of several experts, gathered around a blackboard and *iteratively* solving a problem has several advantages [EM88] among which are:

- **Natural Parallelism:** data can be processed by KSs as soon as they are available. Moreover, the data-flow model associated with this approach is easy to parallelise.
- **Multiple levels of representation:** as we will detail later, Blackboard systems were originally developed to solve signal processing problems [EL80, NFAR88] with several levels of representation. In this architecture, several data formats can cohabit on the BB while being processed by different KSs, simplifying chained analysis.
- **Constant visibility:** as data are constantly pushed to a common data-structure, allowing KSs to interfere with non-finalised, deriving possible solutions.
- **Knowledge sand-boxing:** KSs can be of arbitrary complexity as their internal processing is not visible. They can contribute to the analysis at the only requirement of sharing a representation (data-type) with another KSs.

6.3.1 BlackBoard Architecture

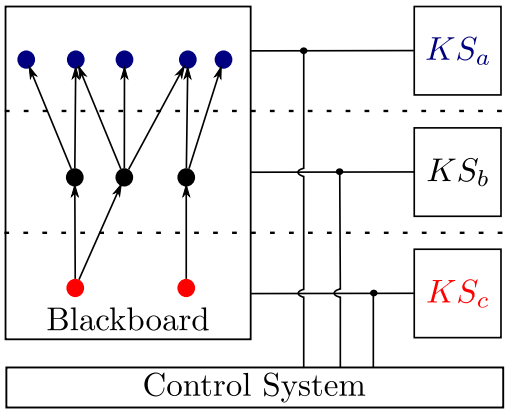


Figure 6.4: *Canonical BlackBoard architecture.*

As presented in Figure 6.4, a canonical BB framework consists in several components enabling *opportunistic reasoning*. The blackboard model being at first conceptual, it has to be

adapted to a computing substrate by adding a control system in charge of triggering data computation [EM88]. Consequently, a Blackboard framework gathers the following components:

- The **BlackBoard (BB)** is a data-structure used as common denominator between KSs. Data are organised hierarchically in different levels and objects are namely identified[3]. Blackboards allowing multiple levels [ET79] with distinct data representations.
- **Knowledge Sources (KSs)** are either procedures or rules, defining how data present on the BB are processed and represented in a domain specific fashion. A KS generally inform the control module about how it can contribute to a given solution, deciding which KS to trigger on a given data. KSs which are only allowed to communicate through the BB.
- The **Control Module** provides a control flow to a blackboard model implementation, reacting to changes on the BB. It is also in charge of ranking KSs contribution to decide which data will be processed next (also reffered to as *focus of attention*).

Consequently, a classical Blackboard work-flow could be described as follows: (1) several KSs are gathered around a BB, as (2) new data become available, the control system decides which KS(s) to trigger (*focus of attention*) in function of an heuristic of their contribution. Then, (3) this analysis can at its turn produce new data and so on. The (4) stop condition is also at the discretion of the control system either through a special condition or simply because the solution has been found. Dealing with parallelism, blackboard systems are good candidates for concurrency [Cor88] as their data-flow nature is inherently parallel. Two types of parallelism can be identified [EM88, DQZ90]: (1) executing several blackboards in parallel [LC83, Wil88] with communications in between systems, approach which can be called a distributed blackboard. Or (2) running several knowledge sources in parallel [NAR90] in order to take advantage of the data-flow parallelism. Approaches which can be combined to build a blackboard which is both parallel and distributed [Sch86] with the drawback of preventing global visibility of data (due to memory scattering).

6.4 Data Management

The exponential computing power growth led to a large increase in the amount of data that is being manipulated. This section presents some data management approaches in terms of storage, representation and processing. We start with the file-based approach which is the norm in the HPC context before shifting toward key-value data-stores which are now propelling Internet largest websites. In a second time, we present data analysis methods associated with key value data-stores before finishing with TBONS which are used by some HPC tools for both reduction and control purposes.

6.4.1 File-Based Approach

A common way of managing simulation output is to store them in a parallel file system such as lustre [BS02], GPFS [SH02] or PVFS [RT00] which are specifically tailored to manage several clients, for example, through the replication of meta-data servers which are the main point

[3] This data representation described in [EM88](p.13) has to be compared with current approaches such as No-IO, No-SQL and Map-Reduce which favour key-value data-stores.

of contention in a coherent file-system. However, directly using the POSIX interface to address such parallel file-system is often not recommended as it can lead to poor performance (because of meta-data contention). Possibly leading to file-system instability, possibly impacting the whole machine, despite the availability of meta-data caches in specifically tailored production grade NFS servers [DLL07]. Consequently, in order to run at higher scale the use of parallel I/O libraries is compulsory to reduce the number of files from one per core to one per node (or less). Therefore, such library are in charge of multiplexing data streams through network calls before handing them to the file-system. For example, MPI I/O [MF08,TLG97] embeds this support in the MPI standard and other libraries such as HDF5 [The13] or NetCDF [NET13] define standard ways of managing scientific data in parallel (including internal layout). Another parallel I/O library, which brings less constraints over data-layout is SionLIB [FWP09] which is used by the ScoreP [aMBB+12] framework for parallel traces writing. Consequently, the management of scientific data-set have seen the development of several approaches which for most of them defined both IO abstractions and data-formats, contributing to their complexity. Handling simulation outputs over a file-system abstraction is therefore a challenging task requiring specifically tailored libraries such as, for example, Hercule [BCF+12, Vet13] which defines a meta-model for parallel code coupling purposes.

6.4.2 Key-Value Data-stores

Big-data[4] is a concept which fast-developed in the context of the WEB 2.0 challenges. Indeed, web services such as Facebook, Google, Amazon... now manipulate unprecedented amount of data which are both at the core of their services and economical model. Consequently, most widespread Big-data evolutions came from web actors as means to face their data manipulation challenges, addressing, collection, storage, manipulation, analysis and visualisation problems associated with large data-sets. Firstly, data storage evolved from a transactional one (satisfying the ACID[5] proprieties) to a much simpler one: *NoSQL* data-stores. They rely on a key-value representation and can be viewed as large associative arrays. The first *NoSQL* database was Memcached [Fit04] which allowed temporary data storage in memory with a key/value interface. Approach which has been generalised to persistent databases which were stripped of their SQL[6] interface, greatly decreasing their complexity and therefore, yielding higher performance. Key/value database can also be distributed on several nodes, approach referred to as *sharding* in order to horizontally scale the database (more node) instead of relying on vertical scaling (upgrading nodes) — allowing scaling over commodity machines instead of expensive mainframes. Sharding can be performed on natural data segmentation, indexing, replication, key space splitting with the pitfall of unbalanced data-sets or through consistent hashing which maps a hashing space over distributed nodes. This shift led to the development of several database such as MongoDB [CD10], Facebook's Cassandra [LM09], Google's Bigtable [CDG+08] (cloned in open-source as HBase [Geo11]), Voldemort [Tea13b] (used for example by LinkedIn, an open-source clone of Amazon Dynamo [DHJ+07]), Redis [Tea13a]... Each of them come with its subtleties and advantages while relying on the key-value approach and a sharding method. Several distributed file-systems were designed to scale on commodity hardware for big data usage, including Amazon Simple Storage Service [ws13] (S3), Hadoop Distributed File System [Bor07] (HDFS) (which aims at powering the Hadoop MapReduce framework) or Google File System (GFS) [GGL03].

[4] Sometimes used as a buzzword, but with no doubt highlighting critical problems.
[5] Atomicity Consistency Isolation Durability [Gra81].
[6] Structured Query Language [CB74] see [HM10, Hai12] for underlying concepts.

6.4.3 Distributed Data-Reduction

Derived from the key-value paradigm and inspired from functional programming, the MapReduce [DG08] approach is an algorithm design pattern which as stated by its name consists of mapping a data set to several nodes in order to perform a computation before reducing the result. MapReduce can be used to scatter a problem on a large cluster of commodity machines which for example parse the data contained in chunks (mapping) in order to perform a computation over parsed chunks (reduction), eventually generating a result stored as a file. This simple process is coordinated by a master which is in charge of dispatching jobs while providing fault-tolerance support either through job replication or by restarting failing jobs. MapReduce is a powerful technique describing parallel processing work-flows for large data volumes. One interesting aspect of this approach is that computation is made space independent in the sense that the programmer is prompted to specify primitive with both spatial and temporal aspects, giving parallelism opportunities to the underlying run-time. Indeed, one of parallel programming challenge is scattering data over computing units while following architecture's hierarchy (particularly with a steadily increasing number of cores). Therefore, expressing computation in term of data dependency and spatial operations (thanks to an explicit data scattering scheme or as in MapReduce a shuffle operation) can be an opportunity for simplifying parallel programming models. There are numerous implementations of MapReduce with, for example, Hadoop [Whi12] (which uses Java) and even over MPI [PD11].

6.4.4 Tree-Based Overlay Networks (TBONS)

Tree-Based Overlay Networks or TBONS [APM06] are a scalable structure for control and measure over large cluster of nodes. They take advantage of trees logarithmic complexity in order to connect several processes in a space efficient manner. They can be used to implement high throughput reductions and broadcasts with custom filters. The reduction case opens opportunities for streamed data reduction, processing data through filters at each tree level. In this case, instrumented processes are part of leaf nodes (called back-end) and data are streamed to the front-end (root node) while being processed by several reduction filters. This method is used in the DDT debugger [All13a] for control and program state reduction purposes. Frameworks such as MrNET [RAM03, JBM12] which is part of the Paradyn project [MCC+95] can be used to build efficient TBONS with arbitrary reduction filters, framework which proved its scalability in the STAT debugger [AdSL+09]. TBONs are also used in the Generic Tool Infrastructure (GTI) [HMdS+12] which relies on PNMPI and provides a generic infrastructure to instrument and reduce events. The GTI allows instrumentation, transport and reduction through XML specifications and has been successively used to offload and parallelise MUST's validations. Examples of profiling tools using the TBON paradigm are Periscope [BPG10] and MAP [All13b] (derived from DDT) which operate a tree-based reduction on performance metrics.

Part III

Contribution

CHAPTER 7

MPI Runtime Characterisation

This chapter introduces a tool which aims at characterising the execution substrate from the MPI programming model viewpoint. This tool has been firstly developed to acquire a better understanding of machine constrains and capabilities in prevision of application profiling. Its purpose is to empirically explore MPI based program performance by characterising individual MPI calls in function of various parameters. This process could be described as fingerprinting a given supercomputer in the purpose of generating a reference document allowing developers to objectively assess their programs costs in terms of MPI calls. This chapter firstly describes our characterisation tool followed by excerpt from reports studied through an empirical cost analysis of the MPI interface.

7.1 Tool Architecture

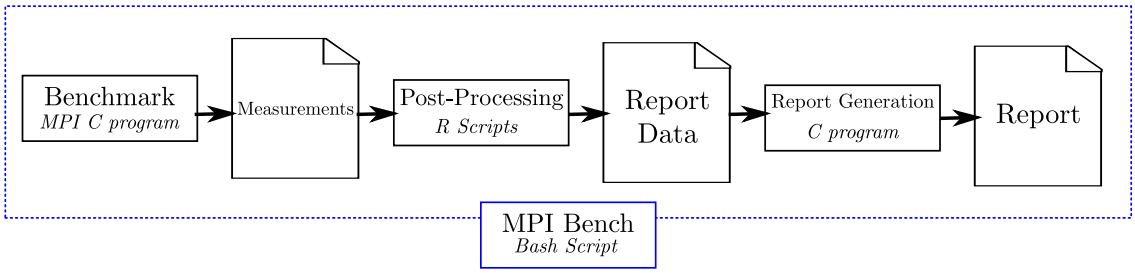


Figure 7.1: *Overview of the architecture of our MPI Benchmarking tool.*

As presented in figure 7.1, our MPI benchmarking tool *MPI_Bench* is based on a bash script which provides a convenient interface for both measurements and report generation. The bench-marking process relies on a simple MPI program which stores several realisation durations for each MPI call using varying processes counts and message sizes. Measurements are collected in an iterative fashion, samples being appended at the end of existing measures. It allows a better test coverage as samples are not necessarily correlated in time as many factors can influence machine load: hour of the day, holidays, automated runs... Once collected measurements are post-processed by R scripts in order to extract for each case common metrics (such as average, minimum, maximum, deviation) while producing associated graphs. These data once processed form a *Report Data bundle* which can also be used to produce comparative reports. Then, this *Report Data bundle* is converted into a latex report which regroups

all measurement with hyperlinks redirecting to associated realisation graphs and probability density functions, for each $\{size, process\ count\}$ combination. Eventually, this report can be compiled as an autonomous PDF file for further reference.

7.2 Measurement Process

The measurement process is carried over by a simple MPI c program which times several realisations for most MPI calls using different $\{size, process\ count\}$ parameters, thus, exploring the possible performance space of the MPI interface. This section details our measurement method for different types of MPI calls.

7.2.1 Point to Points

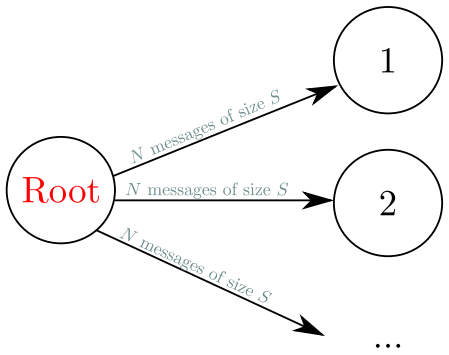


Figure 7.2: *Process sweeping used by our point to point measurement method.*

Send, Ssend, Isend and Issend point to point communications are measured for different message sizes and process counts. As presented in figure 7.2, we rely on a simple sweeping measurement by proceeding with N measurement of size S on several processes. We use this approach to cover most topological cases at both intra-node (see figure 2.8(a)) and inter-node level (see figure 2.9(a)) as latency and bandwidth vary with process layout. Measurement is performed from sender point of view by measuring the send duration. Using this simple approach, we are able to derive minimum, maximum and average time for each message size on a representative sampling of the topology (thanks to sweeping). Moreover, when dealing with asynchronous messages, we also measure the asynchronous window or time for the request to be returned by MPI_Wait during the sending process.

7.2.2 Collectives Operations

As depicted in figure 7.3, nothing prevents some processes to leave the collective operation once they have made their contribution, making timing of such operation less obvious. Therefore, in our approach, we decided to consider each individual collective call viewed from a given process as a realisation of this collective call. Consequently, calling a collective, for example, on 512 processes yields 512 measurements of collectives, thus, leveraging measurements ambiguities. As before, this measurement is performed for various message sizes and number of processes to provide an outlook over the performance space. Measured collectives are: MPI_Bcast, MPI_Reduce, MPI_Allreduce, MPI_Alltoall, MPI_Scatter, MPI_Gather, MPI_Barrier.

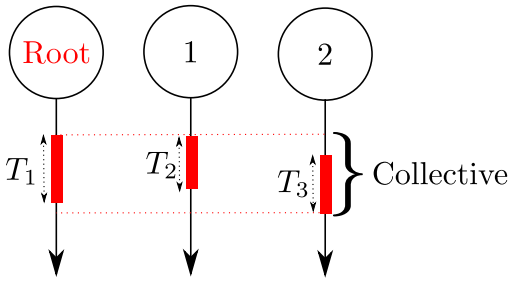


Figure 7.3: *Collective operation measurement method which collects per-process durations.*

7.3 Report Analysis

This sections presents report excerpts and some remarks linked to the measurements we performed on the Tera 100 supercomputer under different conditions. We first cover point to points measurements, emphasising how they reveal the underlying topology. Then, we analyse the influence of machine load over collective operations and conclude that there is an important performance noise which might question performance reproducibility. All measurement were done over MPI Bull which is a derivative of Open MPI provided by machine vendor.

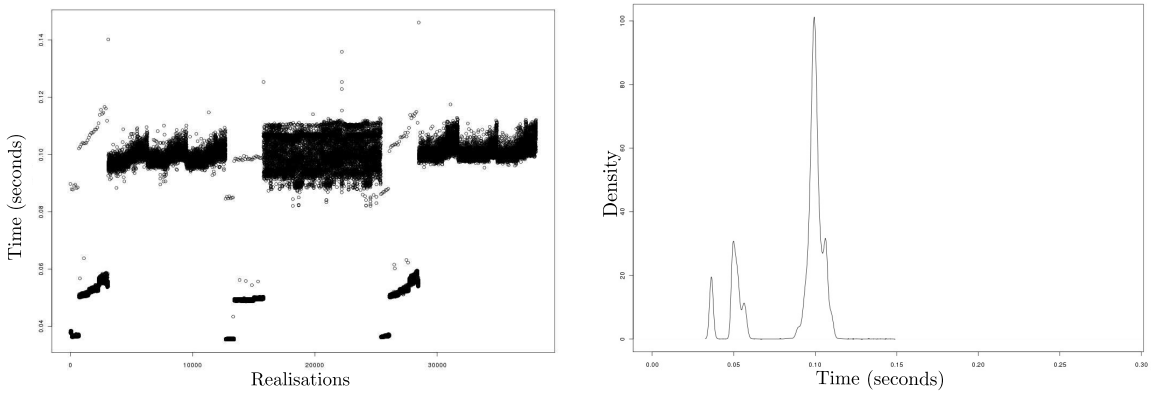
7.3.1 Point to Points

Size (B)	Average	Min	Max	Deviation
4	101 µs	0.146 µs	0.284 s	3.4 ms
8	0.59 µs	0.16 µs	80.7 µs	1.2 µs
16	1.22 µs	0.162 µs	0.266 s	338 µs
32	0.614 µs	0.185 µs	158 µs	1.18 µs
64	0.645 µs	0.199 µs	60.3 µs	1.1 µs
128	0.796 µs	0.201 µs	192 µs	1.12 µs
256	0.981 µs	0.213 µs	0.133 s	141 µs
512	0.832 µs	0.227 µs	180 µs	1.18 µs
Size (KB)	Average	Min	Max	Deviation
1	1.25 µs	0.265 µs	0.131 s	180 µs
2	1.68 µs	0.381 µs	0.125 s	133 µs
4	3.88 µs	1.1 µs	0.13 s	365 µs
8	5.05 µs	1.77 µs	285 µs	10.7 µs
16	28.3 µs	4.7 µs	9.96 ms	89.1 µs
32	40.6 µs	8.11 µs	16.3 ms	25.6 µs
64	64.8 µs	12.7 µs	1.49 ms	17.8 µs
128	107 µs	20.8 µs	10.9 ms	35.7 µs
256	189 µs	39.3 µs	20 ms	72.8 µs
512	347 µs	79.1 µs	18.6 ms	104 µs
Size (MB)	Average	Min	Max	Deviation
1	666 µs	157 µs	14.4 ms	169 µs
2	1.28 ms	312 µs	17.2 ms	334 µs
4	2.45 ms	626 µs	17.7 ms	560 µs
8	4.94 ms	1.25 ms	19.6 ms	1.14 ms
16	9.94 ms	2.5 ms	28 ms	2.25 ms
32	20.3 ms	7.84 ms	0.118 s	3.8 ms
64	39.9 ms	17.7 ms	0.115 s	8.1 ms
128	87.8 ms	35.4 ms	0.146 s	22.7 ms

Figure 7.4: *MPI_Send in function of message size.*

Size (B)	Average	Min	Max	Deviation	Average Window	Min Window	Max Window	Window deviation
4	0.617 µs	0.127 µs	5.19 ms	18.3 µs	100 µs	0.0229 µs	0.133 s	3.39 ms
8	0.47 µs	0.143 µs	45.6 µs	0.397 µs	0.503 µs	0.0247 µs	0.211 s	223 µs
16	0.478 µs	0.145 µs	45.8 µs	0.434 µs	0.269 µs	0.0247 µs	57.9 µs	1.13 µs
32	0.5 µs	0.161 µs	46.1 µs	0.405 µs	0.264 µs	0.0247 µs	151 µs	1.05 µs
64	0.539 µs	0.166 µs	74.2 µs	0.451 µs	0.369 µs	0.0247 µs	88.6 ms	93.8 µs
128	0.626 µs	0.191 µs	141 µs	0.552 µs	0.312 µs	0.0247 µs	55.4 µs	0.819 µs
256	0.635 µs	0.196 µs	191 µs	0.482 µs	0.312 µs	0.0247 µs	51.3 µs	0.827 µs
512	0.637 µs	0.201 µs	59.9 µs	0.453 µs	0.314 µs	0.0247 µs	58.8 µs	0.881 µs
Size (KB)	Average	Min	Max	Deviation	Average Window	Min Window	Max Window	Window deviation
1	0.741 µs	0.244 µs	80.4 µs	0.461 µs	0.341 µs	0.0247 µs	84.3 µs	0.993 µs
2	1.24 µs	0.341 µs	287 µs	0.926 µs	0.367 µs	0.0247 µs	54 µs	1.17 µs
4	1.75 µs	0.515 µs	90.3 µs	0.527 µs	1.01 µs	0.0247 µs	96.4 ms	102 µs
8	2.58 µs	0.519 µs	127 µs	0.822 µs	2.29 µs	0.0247 µs	0.124 s	132 µs
16	3.09 µs	0.559 µs	69 µs	0.982 µs	23.5 µs	4.14 µs	9.43 ms	87.9 µs
32	3.14 µs	0.559 µs	82.4 µs	1.01 µs	36.2 µs	7.62 µs	11.7 ms	22.7 µs
64	3.17 µs	0.584 µs	60.2 µs	0.975 µs	60.4 µs	12.2 µs	8.7 ms	19.3 µs
128	3.26 µs	0.558 µs	305 µs	1.17 µs	103 µs	20.2 µs	17.5 ms	35.4 µs
256	3.28 µs	0.575 µs	119 µs	0.957 µs	182 µs	38.7 µs	5.05 ms	45.7 µs
512	3.4 µs	0.565 µs	63.7 µs	1.06 µs	341 µs	78.6 µs	17.7 ms	86.6 µs
Size (MB)	Average	Min	Max	Deviation	Average Window	Min Window	Max Window	Window deviation
1	3.5 µs	0.595 µs	43.2 µs	1.16 µs	658 µs	156 µs	5.23 ms	153 µs
2	4.47 µs	0.625 µs	45.2 µs	1.52 µs	1.27 ms	311 µs	17 ms	308 µs
4	4.2 µs	0.635 µs	63.9 µs	1.71 µs	2.29 ms	625 µs	13.6 ms	668 µs
8	4.5 µs	0.649 µs	895 µs	4.93 µs	4.59 ms	1.25 ms	21.5 ms	1.36 ms
16	4.67 µs	0.709 µs	44 µs	1.85 µs	9.24 ms	2.5 ms	27.9 ms	2.76 ms
32	5.07 µs	1.67 µs	297 µs	2.17 µs	19.1 ms	8.07 ms	57.8 ms	4.64 ms
64	5.13 µs	1.82 µs	56.2 µs	1.68 µs	38.7 ms	17.6 ms	0.114 s	8.83 ms
128	8.08 µs	1.87 µs	1.76 ms	33.5 µs	87.3 ms	35.3 ms	0.215 s	22.6 ms

Figure 7.5: *MPI_Isend in function of message size.*



(a) Realisation graph.

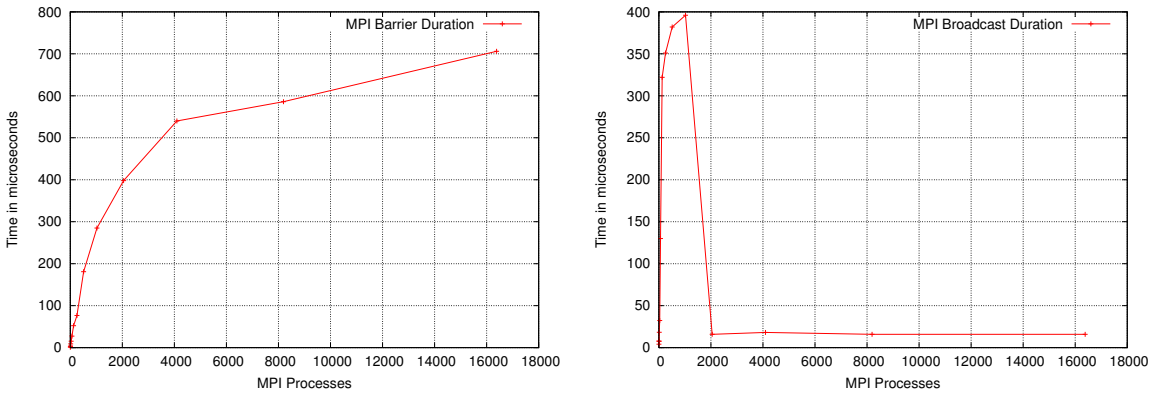
(b) Time distribution.

Figure 7.6: *Sample graph outputs for MPI_Send at 128 MB (for three successive runs).*

Figures 7.4 and 7.5 present sample measurement outputs as generated by MPI Bench. It can be seen in both cases that first communications are more expensive (see 4 Bytes measures) as they also account for queue pair constructions which requires several extra control messages. It can be seen in figure 7.4 that the send time of blocking messages matches the asynchronous window of non-blocking ones in figure 7.5. Dealing with graphs, figure 7.6(a) presents 40 000 realisations of 128 MB MPI_Sends, whereas, figure 7.6(b) shows the associated duration distribution. Figure 7.6(a) consists in three successive measurements at different moments of the day. Machine's topology is clearly visible as the batch manager places processes with contiguous ranks close to each other on the actual topology. Thus, they are first gathered on a socket, then on remote sockets and eventually on another node with the same pattern. This measurement was done over 128 processes, or 4 nodes which are clearly visible on each of the three measures. As shown in figure 7.6(b), this measure reveals three duration classes

which match (1) intra-socket, (2) inter-socket and (3) inter-node communications. Moreover, the NUOIOA [Mor11] effect is visible in measurements 1 and 3 where the socket which is closer to the network interface has better performances than remote ones, yielding this saw-toothed shape for remote node communications. If we look at the second measurement of figure 7.6(a), it can be seen that it is much more noisy than the two other ones, at the point of hiding NUOIA effects, showing that machine load has an impact on performance, and thus, that at some point the performances of a given program might not be always reproducible.

7.3.2 Collective Operations



(a) MPI_Barrier.

(b) MPI_Bcast 8 KB.

Figure 7.7: *Sample graph outputs for MPI collectives.*

Our benchmark tool also measures MPI collective operations performance. In complement of tables similar to the one we presented for points to points, several graphs are generated in purpose of visualising those measurements for each parameter combination — leading to a relatively large 319 page report. As presented in figure 7.7, if we look at those graphs for collective operations, it is possible to get some insight on machine performance. As Tera 100 is constantly used at ≈90%, larger scale, collective related measurements had to be performed during a maintenance, over an empty machine. Figure 7.7(b) which presents the evolution of MPI_Bcast duration for a 8 KB size in function of the number of processes is an illustration of machine load impact over performance. Indeed, if we look a the first measurements until 2048 processes, we find the expected logarithmic shape, but passed this value, when running on the empty supercomputer, there is a significant decrease of collective time, emphasising performance machine load impact. However, if we look a figure 7.7(a) which presents MPI_Barrier performance and has been done in the same conditions, we can see that there is no performance improvement between empty and full machine. This behaviour can be explained by looking at the collective implementation itself, as the two collectives of figure 7.7 does not incur the same network traffic. Indeed, MPI_Barriers are commonly implemented over small messages or RDMAS [KLW+03], whereas, an MPI_Bcast usually relies on a tree which replicates data in order to accelerate data dissemination. Consequently, the difference of behaviour observed in figure 7.7 can be explained by the higher bandwidth budget when the machine is empty, bandwidth which is not required to perform an MPI_Barrier.

7.4 Summary

We have presented a simple MPI bench-marking tool which can be used to measure common MPI calls in standard conditions for various process count and message sizes. This tool generates latex reports including performance tables (as those of Figure 7.4 and 7.5), realisation graphs (Figure 7.6(a)) and their associated distribution (Figure 7.6(b)). Those measurement shown that messages are subject to network noise and that performance is clearly topology dependent by highlighting tree topological levels (see distribution peaks in Figure 7.6(b)). Dealing with collective operations, our measurement process which for practical reasons was partially done on an empty machine revealed the critical impact of network load over collective performance (Figure 7.7(b)). This poses the question of performance reproducibility as machine load is constantly fluctuating [Vet13](p. 70, figure 4.19). Moreover, as processes layout differs at each run, topology related costs are also responsible from unpredictable performance variations. Consequently, those measurements emphasised the non-deterministic aspect of performance on large clusters, stressing the need for a better understanding of those effects. Nonetheless, despite those topology and load effects, Infiniband QDR does furnish high performances, for example, sending small messages microsecond range and larger ones (128 MB) in one tenth of a second, providing application with efficient networking capabilities.

CHAPTER 8

Timestamp Synchronisation

This section presents, justifies and analyses the clock synchronisation algorithm which is used by the successive versions of our distributed tracing library. Related work in terms of clock synchronisation are described in section 6.2. This chapter develops a synchronisation algorithm which relies on a self-testing feedback loop assuming that if correctly synchronised, two successive synchronisations shall have comparable offsets (because of samples' normal distribution). In this purpose, we first detail the synchronisation process between two clocks before extending it to n clocks. Then, we analyse the error induced by our synchronisation approach before concluding on its usefulness for tracing purposes.

8.1 Synchronisation Principle

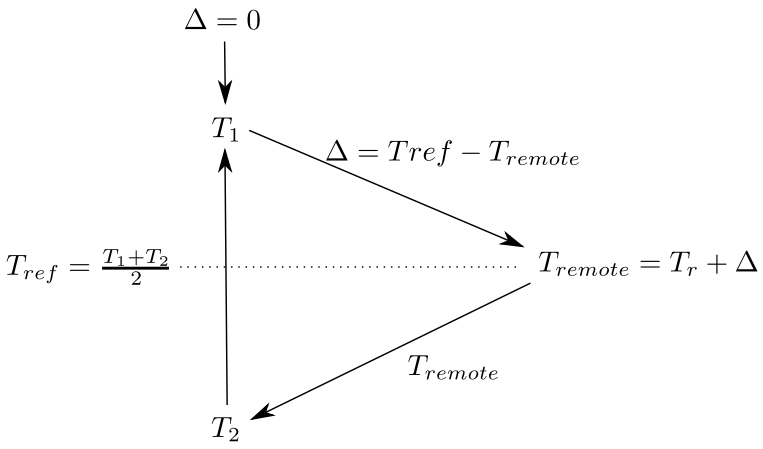


Figure 8.1: *Synchronisation process using a feedback loop.*

Our approach relies a feedback loop which is a common way of synchronising distributed clocks (see section 6.2). Figure 8.1 presents the synchronisation process. The *remote* process or P_r sends to the *source* process or P_s its local time corrected by the Δ computed by P_s (initially $\Delta = 0$). This synchronisation relies on the fact that the communication time is relatively constant in both directions between P_r and P_s, it can therefore be assumed that upon reception, the time in P_r is equal to $T_{ref} = \frac{T_1+T_2}{2}$ with T_1 and T_2 times respectively when sending and receiving to and from P_r. The network being a shared resource, it is naturally subject to noise,

therefore, each Δ is computed as an averaged value from 400 round-trips around the feedback loop. Moreover, the synchronisation operation can be resumed up to ten times, if P_s fails to predict on a second round-trip average T_{remote}, or $t_{lim} <= \Delta$ with t_{lim} an empirical threshold (computed as 4σ in section 8.4.1) determining an expected synchronisation range, this process validates that the synchronisation is possible despite network noise. If it fails, a wider t_{lim} has to be manually set in order to relax the expected accuracy.

8.2 Distributed Synchronisation

Naturally, when dealing with several thousand clocks, the method described in previous section has to be taken further in order to synchronise more than two processes. As we will show this process involves a trade off between synchronisation accuracy and its parallelism. This section describes the effects of using common topologies in order to synchronise N processes. These topologies will be analysed in terms of synchronisation cost and accuracy, result which justify the synchronisation algorithm retained in our implementation.

8.2.1 Notations and Methodology

In the rest of this section we will analyse topology impact on the synchronisation process in order to observe two parameters which as we will show are mutually exclusive:

- **Accuracy:** measures the largest absolute difference between two clocks.
- **Parallelism:** how many processes can synchronise themselves concurrently.

Dealing with the *accuracy*, we consider that the error induced by the synchronisation loop (described in 8.1) noted E_{sync} is in the worst case bounded by round-trip time as feedback process guarantees that $T_{remote} \subset [T_1; T_2]$. Indeed, T_{remote} causally resides between T_1 and T_2 from the source point of view. Similarly, T_{remote} is explicitly corrected by Δ to match T_{ref} in in a process taking the round-trip time. We could have expressed the error in term of latency arguing that this loop involves two temporally uncertain steps which are communications going back and forth, therefore costing two times the latency. But this value is not measurable with a satisfying accuracy on two distinct clocks, having therefore to be approximated as half the round-trip in order to be measured by a single clock. More formally, as the communication latencies involved in the process, are symmetric T_{ref} can be accurately computed as $T_{ref} = \frac{T_1+T_2}{2} = \frac{(T_r-l)+(T_r+l)}{2}$ with l the network latency and T_r the reception time as $T_r = T_1 + l = T_2 - l$. Therefore, we have $T_{ref} = T_r + E_{sync}$ accounting for two time the latency error which can be assumed to be the round-trip error (E_{rt}) such as $E_{sync} = 2E_{latency} = E_{rt}$. Synchronisation error is consequently measurable as the round-trip error, value which as we will see can be described statistically.

As far as parallelism is concerned, we focus on how fast N clocks can be synchronised. In order to limit the noise during the measurement process, the synchronisation process is an exclusive operation between two processes (each with their own clock). Therefore, maximum parallelism is $P = \frac{N}{2}$ with N the number of distinct clocks (we assume that N is also the number of distributed processes). Doing parallel clock synchronisation also supposes that we transitively synchronise clocks such as if we have $T_1 = T_2 + E_{sync_{12}}$ and $T_2 = T_3 + E_{sync_{23}}$ we get $T_1 = T_3 + E_{sync_{13}}$ with $E_{sync_{23}} = E_{sync_{12}} + E_{sync_{23}} = 2E_{sync}$. Errors are therefore additive and

directly linked the the number of hops between two clocks in the synchronising topology. We can immediately deduce from this observation that the maximum error E_{max} is linked to the maximum distance D_{max} in this topology, feature that we will observe for different topologies in the rest of this section.

The transitive propagation of time-stamps is done across the topology by sending to each node the cumulative offset from the root (process 0). In this case if the topology is for example the tree of figure 8.3, we get $T_1 = T_0 + \Delta_{01}$ and $T_2 = T_1 + \Delta_{12}$, we get $T_2 = T_0 + \Delta_{02}$ with $\Delta_{02} = \Delta_{01} + \Delta_{12}$. For a given topology the cost of this transitive dispatch is analogous to a broadcast which cost is linked to D_{max}.

8.2.2 Centralised Topology

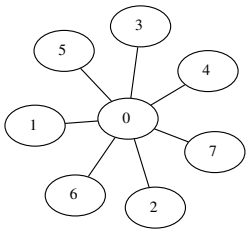


Figure 8.2: *Example of a star connected topology with eight processes.*

From Figure 8.2, it can be immediately seen that $D_{max} = 1$ and therefore, $E_{max} = E_{sync}$ which is the lowest error possible (single hop). However, if we look at the parallelism level, 0 has to successively synchronise itself with every processes, yielding a parallelism of $P = 1$ which is the worst case. Using this topology synchronising N clocks two by two has a cost $c_{sync}(N) = \Theta(Ns)$ with s the cost of a synchronisation. Moreover, as $D_{max} = 1$, the overall synchronisation cost is also $c(N) = \Theta(Ns)$ as every process is connected to the root.

8.2.3 k-tree Topology

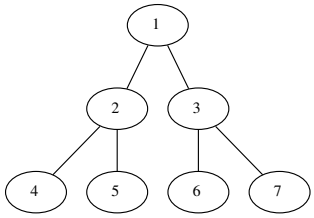


Figure 8.3: *Example of a 2-tree topology with eight processes.*

If we first consider the classical 2-tree of Figure 8.3, maximum distance is equal to tree depth such as $D_{max}(N) = \lfloor \log_2(N) \rfloor$, leading to $E_{max}(N) = \lfloor \log_2(N) \rfloor E_{sync}$. Moreover, by looking at figure 8.4, it can be seen that the level of parallelism depends on the degree of individual nodes. As presented graphically, in Figure 8.4, synchronising every nodes two by two requires

two steps independently from the oddness of tree depth. Indeed, as levels are grouped in pairs we form groups of tree processes (for 2-tree boxes with red dashed lines) which have to perform synchronisation in parallel. However, in order to fully synchronise processes, a second step is required (blue boxes with dashed lines) in order to assign an offset to every edges of the graph. As for a binary tree two synchronisations have to be performed by every group of tree processes and that this process has to be done two times, we get a pairwise synchronisation cost $c_{sync}(N) = \Theta(4s)$ with s the cost of a synchronisation. Therefore in function of $D_{max}(N)$ we get an overall synchronisation cost of $c(N) = \Theta(4s + \lfloor \log_2(N) \rfloor)$.

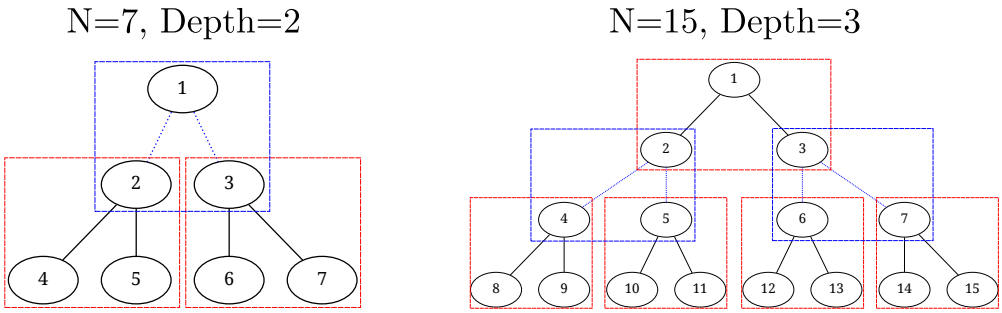


Figure 8.4: *Visual explanation of synchronisation parallelism for a k-tree (k=2) in function of depth's oddness.*

It is also possible to derive from the 2-tree case the generalisation for k-trees. In this case we can define $D_{max}(N) = \lfloor \log_k(N) \rfloor$ and thus, $E_{max}(N, k) = \lfloor \log_k(N) \rfloor E_{sync}$. We can also compute P considering that for a k-tree each process group gathers $k+1$ processes which have to perform k synchronisations, again in two successive steps, yielding a pairwise synchronisation cost $c_{sync}(N, k) = \Theta(2ks)$ with s the cost of a synchronisation. Consequently, we get in function of $D_{max}(N, k)$ the overall synchronisation cost which is $c(N, k) = \Theta(\lfloor \log_k(N) \rfloor + 2ks)$.

8.2.4 Ring Topology

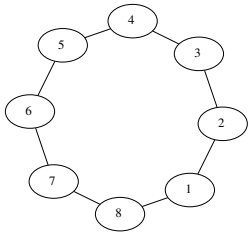


Figure 8.5: *Example of a ring topology with eight processes.*

The ring based topology of Figure 8.5 has a diameter of $D_{max} = \lfloor \frac{N-1}{2} \rfloor$ and therefore leads to a maximal error $E_{max}(N) = \lfloor \frac{N-1}{2} \rfloor E_{sync}$. Dealing with parallelism, as presented in Figure 8.6, each node has to synchronise with two neighbours. Note the particular case of an odd number of processes where one edge (in bold red) is voluntarily ignored in order to gain one synchronisation step, this omission does not prevent broadcasting as there is still a fully synchronised path for both 5 and 4 even without the $5 \leftrightarrow 4$ edge. We can therefore compute the pairwise

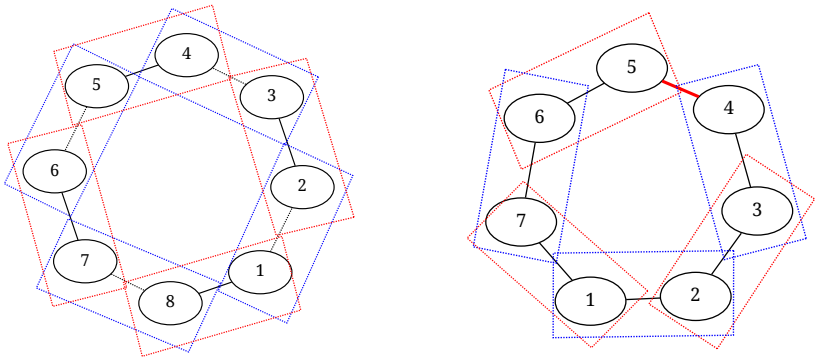


Figure 8.6: *Visual explanation of synchronisation parallelism on a ring in function of N's oddness.*

synchronisation cost as $c_{sync}(N) = \Theta(2s)$ with s the cost of a synchronisation and deduce from D_{max} the overall cost $c(N) = \Theta(2s + \lfloor \frac{N-1}{2} \rfloor)$.

8.2.5 Binomial Tree Topology

A binomial tree of 2^k nodes has a depth of k and is such as the degree of its root is k. A binomial tree of order k noted B_k can be built recursively from two binomial tree of order $k-1$ (which by construction have roots with a degree of $k-1$). It leads to $D_{max} = \log_2(N)$ and therefore to $E_{max}(N) = \log_2(N).E_{sync}$. Dealing with the synchronisation cost, we have as in Figure 8.8, a first step where every node is synchronised with its parent. Then each binomial tree root has to be synchronised with its sub-trees, leaving for every sub-tree roots and particularly for the tree root which has the largest degree $k-1$ synchronisation. This yields a pairwise synchronisation cost of $c_{sync}(N) = \Theta((1+k-1)s) = \Theta(\log_2(N)s)$ with s the cost of a synchronisation. Similarly, we can deduce from D_{max} the overall synchronisation cost $c(N) = \Theta(\lfloor \log_2(N) \rfloor (1+s))$

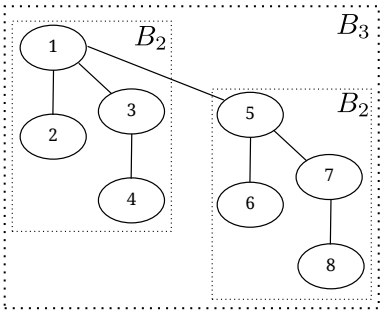


Figure 8.7: *Example of a binomial topology with eight processes.*

8.2.6 Summary

It is possible to derive from previous section Figure 8.9's table which summarises features associated with each topology. Tree based topologies have the lowest synchronisation cost while having a maximum error bounded in $\lfloor \log_2(N) \rfloor E_{sync}$. Particularly, k-tree topologies have the advantage of synchronising themselves in a cost independent of N, being therefore a very

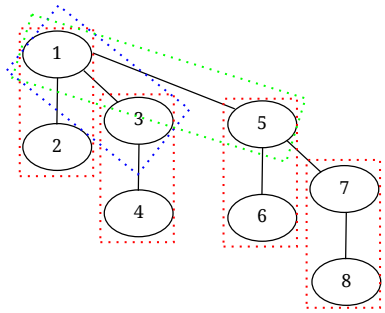


Figure 8.8: *Visual explanation of synchronisation parallelism with a binomial tree (k=3).*

Name	Depth	Max Error	Pairwise Sync. Cost	Transitive Sync. Cost	Global Sync. cost
Centralised	1	E_{sync}	$\Theta(Ns)$	0	$\Theta(Ns)$
Ring	$\lfloor \frac{N-1}{2} \rfloor$	$\lfloor \frac{N-1}{2} \rfloor E_{sync}$	$\Theta(2s)$	$\Theta(\lfloor \frac{N-1}{2} \rfloor)$	$\Theta(2s + \lfloor \frac{N-1}{2} \rfloor)$
Binomial Tree	$\lfloor \log_2(N) \rfloor$	$\lfloor \log_2(N) \rfloor E_{sync}$	$\Theta(\lfloor \log_2(N) \rfloor s)$	$\Theta(\lfloor \log_2(N) \rfloor)$	$\Theta(\lfloor \log_2(N) \rfloor (1+s))$
k-tree	$\lfloor \log_k(N) \rfloor$	$\lfloor \log_k(N) \rfloor E_{sync}$	$\Theta(2ks)$	$\Theta(\lfloor \log_k(N) \rfloor)$	$\Theta(2ks + \lfloor \log_2(N) \rfloor)$

Figure 8.9: *Summary of synchronisation costs (in increasing order, assuming* $N \rightarrow +\infty$ *) for studied topologies.*

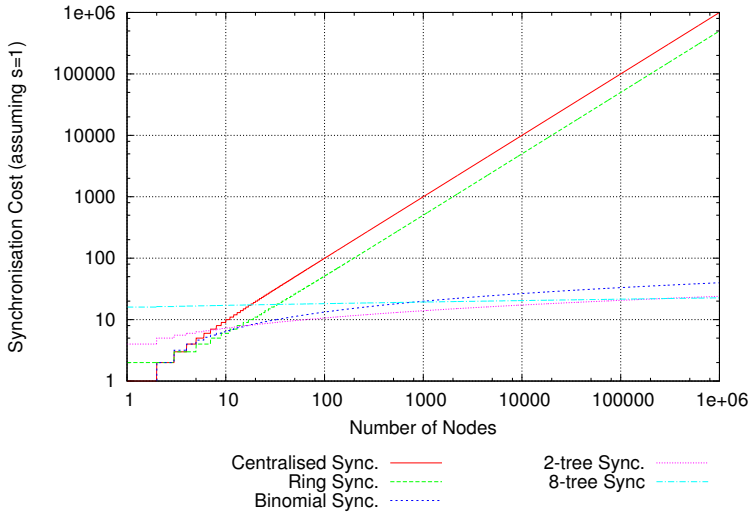


Figure 8.10: *Comparison for studied topologies of synchronisation cost in function node count.*

scalable topology for pairwise synchronisation. Whereas, the binomial tree has a pairwise synchronisation cost which grows in function of N as $\lfloor \log_2(N) \rfloor$ while providing the same E_{max} boundary than a k-tree: $\lfloor \log_2(N) \rfloor$. Dealing with the ring based topology, it is the most parallel one as it synchronises itself in $\Theta(2s)$ however, it has the highest E_{max} which evolves linearly in function of N. Eventually, the star based topology has the lowest error achievable E_{sync} with the highest synchronisation cost which is linear in N. From these observations, a k-tree seems to be the most profitable choice as it has the best scalability while bounding the error in a logarithmic fashion. However, as we will show in next section, the final choice between a k-tree and a binomial tree still depends on another criterion that we propose to analyse in

next section.

8.3 Depth Distribution in 2-trees and Binomial Trees

Before choosing between a k-tree and a binomial tree, another aspect has to be analysed. Although they both share the same error bound and that the k-tree is more scalable, making it from this basis the best choice, a special care has to be taken dealing with their node depth distribution which are drastically different. This section proposes to compute the distributions of nodes among the possible depth for both topologies in purpose of motivating our final choice for the binomial tree.

8.3.1 Notations and Methodology

The node distribution over levels can be described by a function $P(l|\omega)$ which gives the probability of randomly picking a node at level l according to the probability distribution function (PDF) ω. This function can be computed as $P(l|\omega) = \frac{Num(l)}{N}$ with $Num(l)$ the number of nodes at level l with $l \subseteq [0; D_{max}]$ and D the depth. From this function it is possible to derive the cumulative distribution function (CDF) of processes which is defined as $\phi(l, \omega) = \sum_{i<l} P(i|\omega)$, function which gives the cumulative proportion of processes at a depth lower than l according to PDF ω. Ideally, in order to minimise the average error, most processes shall have a low depth, highest depth remaining marginal. However, the gain in parallelism is directly linked to synchronisation processes spatial scattering and therefore to higher errors. In this section we will show that the binomial tree allows a more favourable distribution than the binary tree, justifying its use. In this purpose, we will computes the PDF ans CDF for both cases.

8.3.2 2-tree

By definition the number of node at a given level l in a 2-tree is $Num(l) = 2^l$. And the total number of nodes is $N = 2^{D+1} - 1$ with D the tree depth such as $D_{max} = \lfloor \log_2(N) \rfloor$. We can therefore derive $P(l|\omega_{2tree}) = \frac{2^l}{N}$ with $l \subseteq [0; \lfloor \log_2(N) \rfloor]$. The propriety of having probabilities which sum to 1 can be derived from the identity $\sum_{x<n} 2^x = 2^{n+1} - 1$ or $\frac{1}{2^{n+1}-1} \sum_{x<n} 2^x = 1$ which immediately yields $\sum_{\mathfrak{N}} P(l|\omega_{2tree}) = 1$.

Figure 8.11 presents both $P(l|\omega_{2tree})$ and $\phi(l, \omega_{2tree})$. It can be seen that 50% of nodes are on the last layer, 75% on the two last and 87.5% on the three last layers. This distribution is invariant relatively to N as it originates from the way the tree is built in successive powers of two (the following layer is always two time larger than the current). From this observation, we can conclude that in a 2-tree nodes are generally far from the root and that 50% of nodes have an error matching the upper bound in $\lfloor \log_2(N) \rfloor E_{sync}$.

8.3.3 Binomial Tree

By definition, the number of node at level l in a binomial tree B_k is the binomial coefficient $Num(l, k) = \binom{k}{l}$ with k the order of the tree and l the depth of the node. Moreover, the number of nodes in a binomial tree of order k is $N(k) = 2^k$ we can therefore deduce from the factorial notation of the binomial coefficient: $P(l|\omega_{binom}) = \frac{1}{N}\binom{k}{l} = \frac{k!}{l!2^k(k-l)!}$. Moreover, we

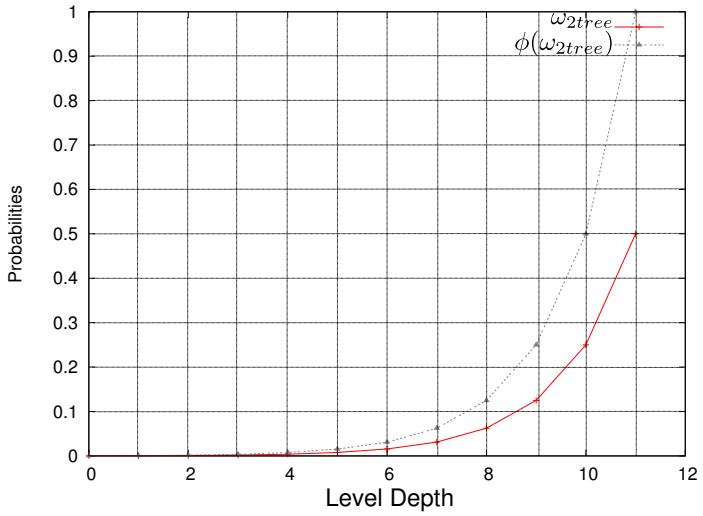


Figure 8.11: *Probability density function* ω_{2tree} *and cumulative density function* $\phi(\omega_{2tree})$ *for node depth in a 2-tree of* $2^{12} - 1 = 4095$ *nodes.*

can guarantee that the sum of all probabilities gives 1 with the identity $\sum_{l=0}^{k} \binom{k}{l} = 2^k$ which immediately gives $\sum_{\mathfrak{R}} P(l|\omega_{binom}) = \frac{1}{N} \sum_{l=0}^{k} \binom{k}{l} = 1$.

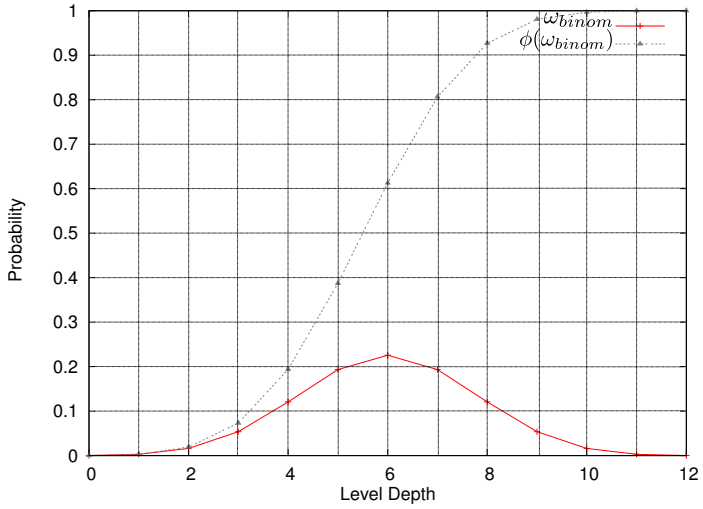


Figure 8.12: *Probability density function* ω_{binom} *and cumulative density function* $\phi(\omega_{binom})$ *for node depth in a binomial tree* B_12 *of* $2^{12} = 4096$ *nodes.*

As presented in Figure 8.12 the depth distribution in a binomial tree is very different from the one of a 2-tree (Figure 8.11). In the binomial case, the distribution is symmetric (as the binomial coefficient are in the Pascal triangle) with the propriety of being evenly distributed around $\frac{D_{max}}{2}$ with D_{max} the binomial tree depth. Consequently, half clocks have an error lower than $\frac{k}{2}E_{sync}$, the other half having an higher error which is nonetheless bounded by $\log_2(N)E_{sync} = kE_{sync}$. Therefore, the binomial tree provides in average a more accurate syn-

chronisation than the 2-tree, moreover, this error balancing can be described as more "natural" as it is somehow equivalent to having every node synchronised with a "virtual" clock located at the level $\frac{k}{2}$ with a random error following a binomial distribution and bounded by $\frac{k}{2}E_{sync}$.

8.3.4 Summary

The more regular depth distribution, and in extension error distribution induced by the binomial topology makes it preferable than a k-tree based topology (exemplified here with a 2-tree) which by construction has more nodes on its deeper levels which have the largest error bound. Moreover, as presented in Figure 8.10, the cost difference between the two approaches is logarithmic $s(\lfloor \log_2(N) \rfloor - 2k)$ and remains acceptable at higher scale in comparison with the accuracy gains it provides. For these reasons our synchronisation relies on a binomial tree to provide our tracing library with a scalable and accurate global clock with an error bounded in $\lfloor \log_2(N) \rfloor E_{sync}$. The following section will study E_{sync} in order to derive a probabilistic definition of E_{max} which fully characterises our synchronisation method accuracy.

8.4 Study of Synchronisation Error Propagation

This sections aims at statistically describing E_{sync} which is the round-trip error and in extension describing E_{max} which is the synchronisation error which can be modelled as $D_{max}E_{sync}$ with D_{max} the depth of the tree and therefore as a sum of D_{max} independent probabilities. In this purpose, this section first derives the PDF of E_{sync} from an empirical measurement. Then, we derive the probabilistic distribution of E_{max}, the overall synchronisation error.

8.4.1 Round-trip Error Distribution

Figure 8.13 presents round-trip latency probability densities for 1.10^9 events, right column shows the unaltered densities for various averaging factors, whereas in left column probabilities have been cut off at a 5.10^{-3} probability. This "long tail" of low probabilities events going to higher latencies has been observed by Cristian in [Cri89], it depicts unpredictable network jitters which can delay the round trip latency of several orders of magnitude. In left column, we empirically thresholded latencies at a 5.10^{-3} probability, removing the "long tail" and simplifying the error distribution. Several heuristics have been proposed to perform this thresholding such as keeping only the lowest time [Cri89, GZ83] or filtering samples as in NTP minimum filter algorithm [Mil91]. As mentioned in section 8.1, our approach also uses a filtering method as once synchronised (400 averaged round-trips), the operation is performed a second time with a new offset expected to be in the 4σ range (99.99 % confidence interval for a normal law), if not, the first measurement was erroneous, then the process starts again, up to ten times. This method relies on the fact that on high performance networks, abnormal times are less probable and more variable than normal ones (long tail versus bell shaped distribution, see Figure 8.13 right column), therefore, the probability of having two erroneous measurements falling in the same 4σ interval is very low, compared to the one of falling in the 4σ of the bell shaped distribution (left column) which accounts for most probable issues.

Dealing with the normality assessment, looking at Figure 8.14 and assuming each round-trip mean as independent experiment, (1) all those distribution seems to be empirically Gaussian and (2) the standard deviation decreases with the number of averaging passes. Moreover,

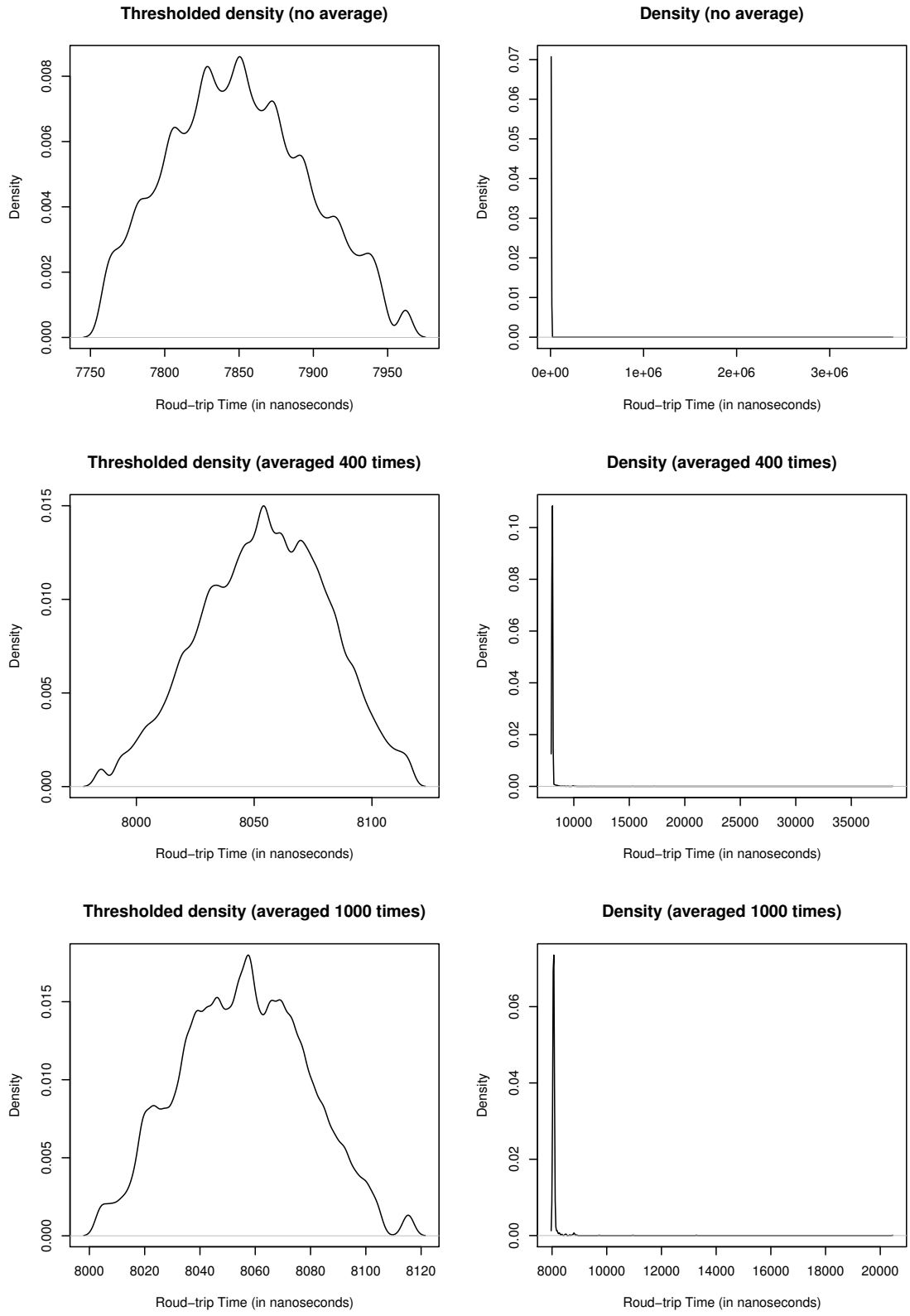


Figure 8.13: *Empirical probability density functions (PDF) for varying averaging factors derived from a set of* 1.10^9 *round-trips on the Tera100 supercomputer either thesholded or not at a* 5.10^{-3} *probability.*

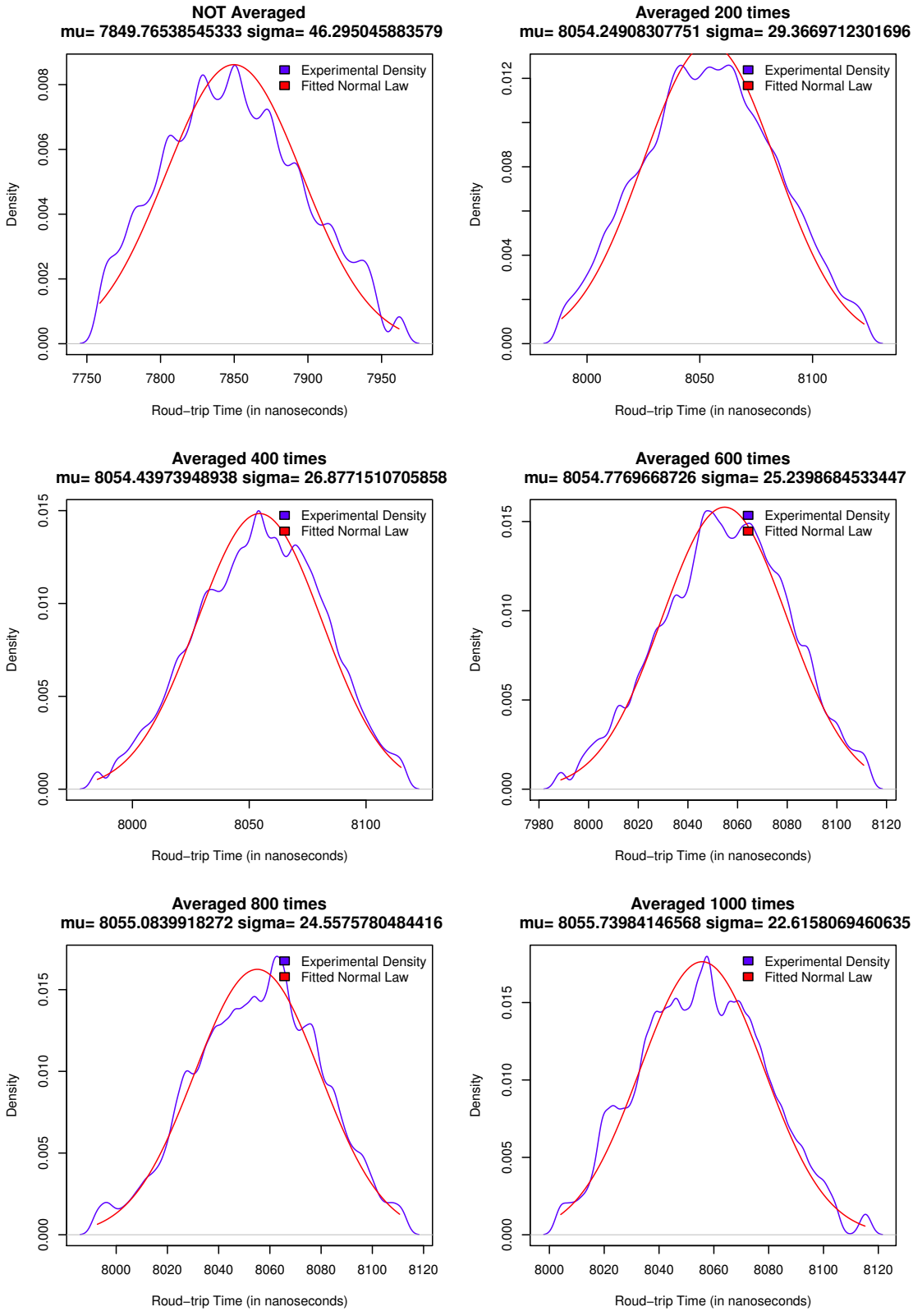


Figure 8.14: *Empirical thresholded (at* 5.10^{-3}*) PDF for various averaging factors which can be fairly approximated by a normal law.*

as presented in Figure 8.15, (3) the quantile to quantile plot (dots) match the one of a normal distribution (line) with only a small mismatch on boundaries (tail effect). Therefore, this measurement can reasonably be seen as obeying the central limit theorem (CLT) which states that under certain condition the mean of of a large number of random variables will converge to a normal distribution.

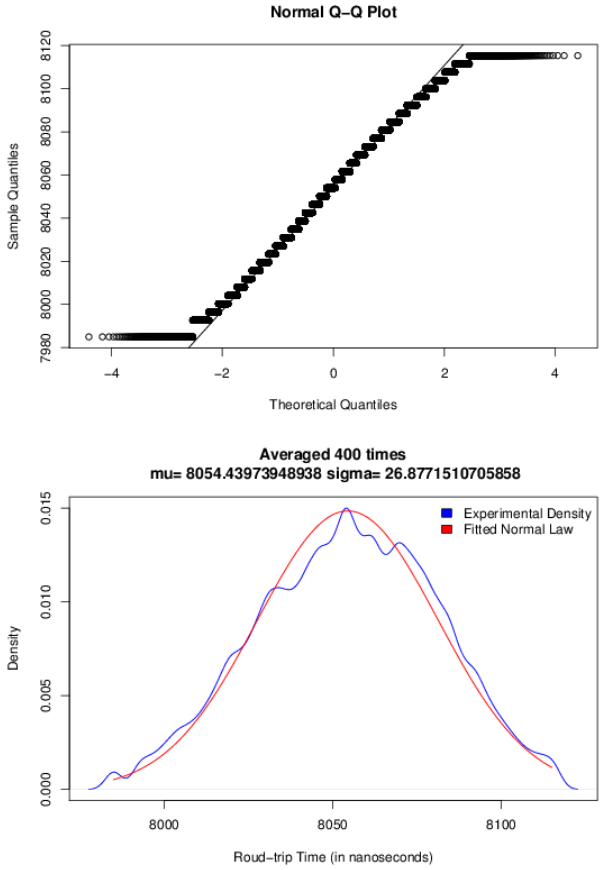


Figure 8.15: *Quantile to Quantile plot (Q-Q plot) normality test for the thresholded PDF associated with measurements averaged 400 times.*

As aforementioned and presented in Figure 8.15, we retained the 400 average value which is a trade off between the synchronisation cost (we noted s) and the attainable precision. As shown in Figure 8.14, following averaging does not provides as much gain relatively to the standard deviation. Moreover, as depicted by the right column of Figure 8.13, the "tail" naturally decreases with the number of averaging passes, "tail" which is required to be long compared to 4σ in order to make the offset prediction test reliable. Consequently, we empirically retained the 400 average measurement for our synchronisation process to match network performances. Although, this setup is specific to high performance networks, fixing the t_{lim} to an higher value is sufficient to adapt this process to other network with lower performance. As the error is depicted by the jitter around the average value, we will now use the following approximation $E_{sync} \sim \mathcal{N}(0, \sigma^2_{sync})$ with E_{sync} the synchronisation error and σ^2_{sync} the variance for a 400 averaging factor such as $\sigma_{sync} = 26.877$ on a nanosecond scale.

8.4.2 Error Propagation

Each pairwise synchronisation on a path used to propagate synchronisation offsets can be viewed as an independent experiment, allowing the probability density functions to be summed to describe $\omega_{E(n)}$, the error probability density function after n hops. The sum of two random variable is their convolution, we note $\omega_{i+j} = \omega_i \circledast \omega_j$ the convolution to two densities and $\omega_{i_n} = \circledast^n \omega_i$ the repetitive self-convolution of a density. In our particular case each pairwise synchronisation obeys the same law, yielding after n steps the following probability density function ω_{E_n}:

$$\begin{aligned}\omega_{E_n} &= \underbrace{\omega_{E_{sync}} + \omega_{E_{sync}} + ... + \omega_{E_{sync}}}_{n} \\ \omega_{E_n} &= \circledast^n \omega_{E_{sync}}\end{aligned}$$

We can therefore derive $\omega_{E_{max}}$ the density at maximum depth D_{max} which is equal to $E_{D_{max}}$:

$$\omega_{E_{max}} = \omega_{E_{D_{max}}} = \circledast^{D_{max}} \omega_{E_{sync}}$$

Moreover, we shown that ωE_{sync} can be approximated by a normal distribution thus $E_{sync} \sim \mathcal{N}(\mu_{sync}, \sigma^2_{sync})$ with μ_{sync} the average round-trip error which is equal to 0 and σ^2_{sync} the round-trip error variance. Moreover, we have by definition:

$$\mathcal{N}(\mu_1, \sigma_1^2) \circledast \mathcal{N}(\mu_2, \sigma_2^2) = \mathcal{N}(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$$

Allowing a simple formulation of ω_{E_n}:

$$\begin{aligned}\omega_{E_n} &= \circledast^n \mathcal{N}(\mu_{sync}, \sigma^2_{sync}) \\ \omega_{E_n} &= \mathcal{N}(\sum^n \mu_{sync}, \sum^n \sigma^2_{sync}) \\ \omega_{E_n} &= \mathcal{N}(n\mu_{sync}, n\sigma^2_{sync}) \qquad (8.1)\end{aligned}$$

Consequently, we have $\omega_{E_{max}}$ such as:

$$\omega_{E_{max}} = \omega_{E_{D_{max}}} = \mathcal{N}(D_{max}\mu_{sync}, D_{max}\sigma^2_{sync})$$

One propriety of the normal distribution being that 99.99% of samples are less than 4σ away from the mean, we will consider that the maximum error is $|E_{max}| = 4\sigma$. It is moreover possible to express σ in function of the number of hops n from equation 8.1:

$$\begin{cases} \sigma^2(n=0) & = 0 \\ \sigma^2(n+1) & = \sigma^2(n) + \sigma^2_{sync} \end{cases}$$

$$\begin{aligned} \sigma^2(n) &= n\sigma^2_{sync} \\ \sigma(n) &= \sigma_{sync}\sqrt{n} \end{aligned} \tag{8.2}$$

It is therefore possible to compute the error $|E(n)|$ at level n and the maximum error $|E_{max}| = |E(D_{max})|$ with D_{max} the maximum number of hops (in our case D_{max} is the tree depth) using the aforementioned 4σ approximation:

$$|E(n)| = 4\sigma(n)$$

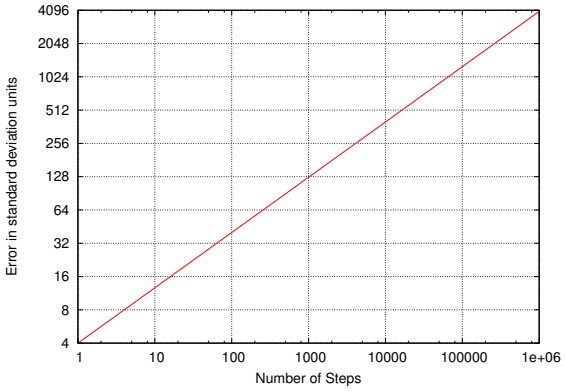
$$|E(n)| = 4[\sigma_{sync}\sqrt{n}] \tag{8.3}$$

$$|E_{max}| = 4[\sigma_{sync}\sqrt{D_{max}}] \tag{8.4}$$

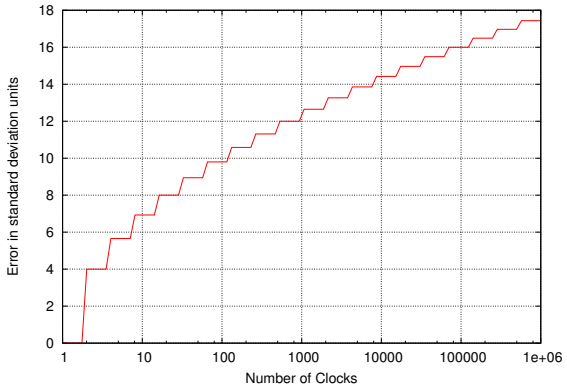
Equation 8.2 and 8.3 respectively show a squared-root increase of standard deviation and the maximum error at 4σ. Also note that the total error once expressed in densities is lower than the trivial bound $D_{max}E_{sync}$ as it is evenly distributed between positive and negative offsets, avoiding its doubling at each step. As presented in Figure 8.16, it is therefore possible to plot the error in function of both the number of steps and the number of clocks, assuming a binomial tree topology. As it can be seen in Figure 8.16(d), the maximum error derived from our empirical measurement of E_{sync} assuming a binomial tree topology and a 400 average factor is lower than $0.5\mu s$ with less than 1.10^6 clocks. Note that this result has to be tempered as it does not account for drift related corrections and has only been derived statistically.

8.5 Summary

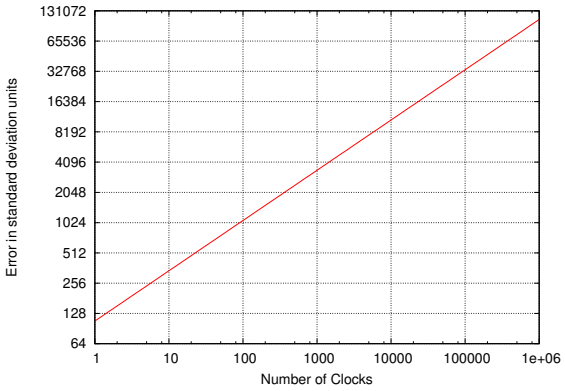
This chapter described the time-synchronisation algorithm used in our instrumentation libraries (Chapter 9 and 10) to provide a coherent source of time. Our synchronisation operates only on offset related error with a statistically derived precision of $0.5\mu s$. However, the method we presented only minimises the offset error and we acknowledge the existence of an unpredictable drift error which fatally influences clock accuracy. Therefore, despite it provides our analysis with a sufficient precision, special care will have to be taken when considering new analysis dependent on time-stamp resolution. Nonetheless, this approximation has been retained because common correction algorithm either have an important computing cost, requiring a full trace replay or can impact the tracing by either causing temporal discontinuities while possibly impacting the execution through repeated synchronisations.



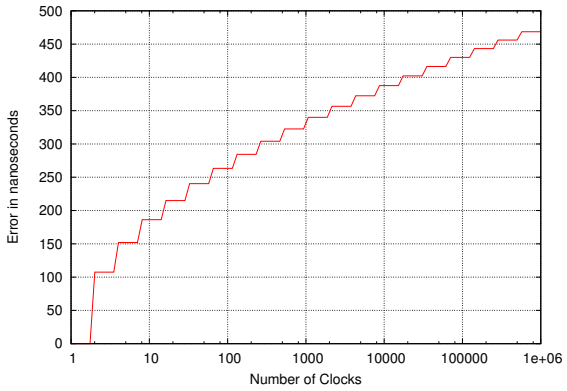
(a) Total error evolution in sigma units (at 4σ) in function of number of steps (tree depth).



(b) Total error evolution in sigma units (at 4σ) in function of number of clocks (binomial tree topology).



(c) Total error evolution in nanoseconds in function of number of steps (tree depth).



(d) Total error evolution in nanoseconds in function of number of clocks (binomial tree topology).

Figure 8.16: *Summary of cumulative error in function of both σ (a and b) and time (c and d).*

We introduced a synchronisation method relying on a classical Cristian algorithm [Cri89] used in a self-testing fashion as it testes its own synchronisation reliability by redoing the synchronisation process a second time with a 4σ expectation, thus, confirming with a high probability the presence of both samples in the bell shaped part of density (see figure 8.13). We justified the use of a binomial tree topology as a trade-off relatively to average diameter, synchronisation and transitive offset propagation costs, emphasising that it yields a more regular error distribution (Binomial $\approx$ Normal versus Exponential), evenly distributing error among processes (see Figure 8.12). Then, we observed the empirical round-trip distribution on the Tera 100 supercomputer and approximated it by a normal law which allowed us to statistically derive our synchronisation method maximum error, showing for less than 1.10^6 processes an expected upper boundary of $0.5\mu s$ using a binomial topology (see Figure 8.16(d)).

Main limitations of our method are (1) we do not take the drift into account assuming that clocks are homogeneous on the supercomputer, (2) we only synchronise at application start-up, avoiding to perform a costly trace rewrite to perform a linear (or more complex) interpolation between end and begin (which could have solved the drift problematic). Moreover,

trace rewriting is not feasible using our on-line approach (see chapter 10) which does not store events anymore. Our time-stamps are also subject to (3) drift variation which can possibly cause clocks to diverge during long runs. However, these limitations have to be mitigated as for the moment no analysis require a distributed resolution approaching the error rate (we generally expect at most milliseconds), accuracy which is in a first approximation guaranteed by our offset based synchronisation.

Because of the aforementioned limitations, our solution, despite satisfactory for the moment can only be a transitive one in the expectancy of a more reliable one. From the arguments we developed in this chapter, we can derive that perfect time-stamp synchronisation is impossible in distributed systems. However, it is possible to capture events causality using logical clocks. A better solution we plan to implement in the future would be to combine the best of both worlds, using an accurate clock to capture intra-node behaviour, a distributed clock (for example using our current synchronisation algorithm) to provide “coarse-grained” event locality and a runtime logical clock to store effective causality. This would tag every events with two time-stamps (local clock can be synchronised globally while keeping its precision) addressing, timing, locality and causality aspects and therefore, taken altogether would provide close to accurate clock capabilities (and slightly more). Capturing causality is important not only to characterise what a program did but also what a program could do by quantifying freedom degrees, analysis which could be useful to derive asynchronism proprieties or parallelism loss. If we quote Fidge [Fid88] in his paper describing the vector clock algorithm: “communication events form "boundaries" that limit the possible interleavings of concurrent events”(p. 57) we have the essence of this approach which aims at considering communications only as a skeleton which constraints the ordering of node local and therefore temporally defined event blocks — approach adopted by Becker et Al. for their extended Controlled Logical Clock (CLC) in grid environments [BGRW13]. Performance optimisation being the identification of programs’ critical path or parallelism inefficiencies and not solely the precise representation of actual event layout which is by nature illusory because of a certain form of uncertainty principle, somehow making modelling more affordable than measure.

CHAPTER 9

Trace Based Approach

This chapter presents the first implementation of our tool which was named after the MPC runtime: the MPC Trace Library. As detailed in section 1.2, it aims at being scalable on current supercomputers in order to be able to characterise simulation codes at their nominal scale. Moreover, it has been designed to comply with MPC which requires an extended topology support and enforces thread-safety. After explaining reasons which motivated the shift to our own trace format, we present the architecture adopted by the MPC Trace Library. Our components presentation will follow the canonical architecture of developer tools (see section 5.1) by first presenting the instrumentation interface, then the coupling mechanism with our trace format and eventually, the analysis side with our dedicated trace reader.

9.1 Limitations of Existing Trace Formats

The first MPC trace library implementation relied on the Open Trace Format (first version, OTF1) [1] [KBB+06] in order to store trace events. However, we faced several limitations which motivated the transition to our own trace format:

- **Thread safety:** one of our requirement was to run over MPC which is a thread-based MPI implementation. Consequently, OTF was loaded once per process and solicited in parallel by several threads. But, as OTF has been designed for process based implementation, we faced problems of thread safety at the file management level, even when creating several "Writers". Limitation which led to the design of our thread-safe file handler (see section 9.4.3).
- **Identifiers:** in OTF1, identifiers are local to each streams, requiring efforts to *unify* event representation at the end of measurement phase, process which led for some tools to a complete trace rewriting. As presented in Section 9.4.1, we decided to rely on a hierarchical which allows identifier direct identifier.
- **Debug support:** one of our requirement being to support faulty programs, we could not rely on a trace format which could not put up with a crashing program as for example, it would loose writing buffers data. Requirement which led to the development of our debug buffers (Section 9.4.4). Moreover, OTF has been designed for temporal traversals, multiplexing events in large time-ordered traces, however, debugging might require the analysis of a single event type, favouring the creation of multiple trace files in purpose of speeding up the traversal during partial readings.

[1] OTF2 [EWG+11] was not available at this time.

- **Storage size:** in OTF1 events are stored in hexadecimal ASCII, approach which as we will show in Section 9.4.6 is not space efficient — motivating the shift to our binary trace format which is more space efficient.
- **Trace processing framework:** upon trace reading, OTF1 does not provide meta-data and job dispatching facilities, requiring a redundant development in each analysis tool, opening and unifying meta-data in a distributed environment. In our approach, we developed the MPC Trace reader which greatly simplifies the processing of large traces by providing a compact interface for meta-data information and distributed processing of MPC traces (see Section 9.5).

All those aspects motivated the development of the MPC trace format instead of relying on the standardised approach provided by the OTF format. If this thesis started today, we would probably have relied on OTF2 which solves most of these limitation and defines an unified framework for several tools, allowing users to take advantage of several tools. An OTF2 back-end is currently under development (next to our online tool which succeeded to this trace-based approach, see Chapter 10) in order to remain in the standards of the profiling community, avoiding redundant developments when standard solutions already exist.

9.2 Proposed Architecture

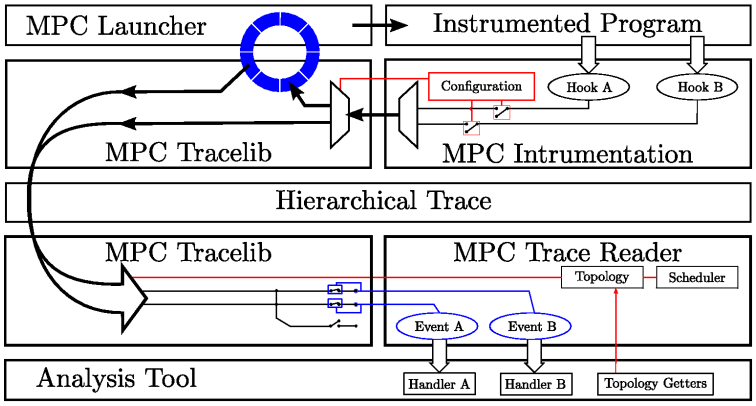


Figure 9.1: *Global architecture of the MPC Trace Library.*

Figure 9.1 presents the global architecture of the MPC Trace Library, it features three main components: the MPC Instrumentation, the MPC Tracelib and the MPC Trace Reader. Two particularities of our approach can already be noticed from this figure: (1) the blue circle represents debug buffers which allow the collection of traces even in case of bad termination and (2) analyses are decoupled from the MPC Trace Library which transparently handles parallelism and topology related information.

9.3 Instrumentation

The MPC Trace Library combines several instrumentation approaches to collect trace events relying on Linux's loader capabilities, the compiler, direct instrumentation and the MPI run-

time. This section details those approach, illustrating each of them with a simple example.

9.3.1 MPI Profiling Interface

A common way of instrumenting an MPI program is to rely on the *MPI profiling interface* or PMPI. This interface, part of the standard [MF08], provides a weak symbol for every MPI functions, allowing their interception.

```
int MPI_Comm_rank( MPI_Comm comm, int *rank) {
  hook_before();
  int ret = PMPI_Comm_rank( comm, rank );
  hook_after();
  return ret;
}
```

Figure 9.2: *Instrumentation of the MPI_Comm_rank function using the MPI profiling interface.*

As presented in figure 9.2, every MPI call (here MPI_Comm_rank) can be instrumented through a redefinition which calls the original function, defined with the same name preceded by 'P'. The MPC Trace Library uses this method for the whole MPI interface. The same process is used for MPC which provides an "MPC_*" interface which also supports profiling (note that MPC also provides the MPI interface, allowing direct compilation of existing MPI programs).

9.3.2 Compiler Level Instrumentation

It is possible to instrument every functions from a program by recompiling it with the *-finstrument-functions* compiler flag which adds a function call before and after every functions (even inlined ones).

```
void __cyg_profile_func_exit (void *this_fn, void *call_site)
__attribute__ ((__no_instrument_function__));
void __cyg_profile_func_enter (void *this_fn, void *call_site)
__attribute__ ((__no_instrument_function__));
void __cyg_profile_func_enter (void *this_fn, void *call_site) {
    /* Entering function */
}
void __cyg_profile_func_exit (void *this_fn, void *call_site) {
     /* Leaving function */
}
```

Figure 9.3: *Handlers called by compiler level instrumentation.*

Figure 9.3 defines the functions which are inserted by the compiler before and after every function calls. Both handlers share the same footprint with the first argument being the function pointer (as defined in symbol table) and the second one the call site. Note that we included their forward declaration with the GCC specific attribute "no_instrument_function" which prevents them from being instrumented if they are part of the target program as it would lead to a recursive call which would inevitably lead to a stack overflow. In order to

be instrumented, each program has to be recompiled (including shared libraries), such instrumentation is therefore limited to user developed programs. Moreover, this process incurs a performance penalty particularly for C++ programs which favour small functions (getters, setters, operators) which are both heavily called and dilated by the instrumentation, leading to non negligible overheads. It would therefore be interesting to filter out small functions in order to limit the overhead, subject which is developed further in appendix A.

9.3.3 Direct Instrumentation

In some particular cases it has been possible to directly instrument target programs. For example, we have been able to add weak symbols (which default to an empty handler) in the MPC runtime in order to capture events of interest. These symbols, when redefined by instrumentation libraries override weak ones.

```
#pragma weak hook_function()
void hook_function()
{
  /* Dummy version */
}

[...]
/* Context of interest */
hook_function();
[...]
```

Figure 9.4: *Definition of a hook in source library.*

Symbol Name	Description
void MPC_Process_hook()	This function is called at process launch, MPI calls are not possible from this state. MPC aware Thread Local Storage (TLS) are not available.
void MPC_Task_hook(int rank)	This function is called for each MPI task, MPI calls are possible from this state. The argument is the rank of current task. MPC aware Thread Local Storage (TLS) are available.

Figure 9.5: *List of initialisation hooks defined in the MPC runtime.*

Figure 9.4 presents the way such hooks are defined in the MPC runtime and figure 9.5 shows the list of functions which are called upon MPC initialisation with their associated constraints. These functions are used to setup the context at process level and for each task (which are running in lightweight threads in MPC). Further details on how to instrument MPC are given in appendix B.

9.3.4 Library Interposition

Linux based systems propose another instrumentation approach which relies on preloading a shared library. This mechanism, allowing the interception of functions defined in shared

libraries is mainly used to instrument calls from the C standard library.

```c
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>

int close(int fd) {
  void *real_close=dlsym(RTLD_NEXT,"close");
  if(!real_close) {
    perror("Failed to retrieve open symbol");
    abort();
  }
  /* Pre hook */
  int ret = ((int (*)(int))real_close)(fd);
  /* Post hook */
  return ret;
}
```

Figure 9.6: *Use of dynamic library preloading for library interposition.*

As presented in figure 9.6, by using the dynamic linking loader it is possible to (1) preload a library which overloads a symbol. (2) Look-up the original symbol through the dlsym function with the linux specific RTLD_NEXT flag. Naturally, an optimised version should not search for the symbol at each call through dlsym. Our implementation in the MPC Trace Library relies on a cache of symbols initialised to NULL and looked up only upon first call.

9.3.5 Instrumentation Summary

Instrumented Component	Method	Description
C Standard library	Library Interposition	**Partial** instrumentation focused on stdio.h and string.h
Pthread library	Library Interposition	**Partial** instrumentation including thread creation, destruction and basic locks
MPI and MPC Interfaces	PMPI and PMPC	**Complete** instrumentation of **MPI 1.3** norm
Function calls	Compiler	Compiler based instrumentation with filtering (as described in appendix A)
Manual instrumentation	Manual	Support for manual instrumentation of functions, timestamped “printf” and value tracking

Figure 9.7: *Summary of instrumented functions in the MPC Trace Library.*

As presented in figure 9.7, the MPC Trace Library supports most common events ranging from PThread to MPI. Each instrumentation source is configurable via a configuration file which defines which events are instrumented for a given execution.

9.4 Trace Library

This section presents the trace management interface which is the main component of the MPC Trace Library. After detailing and motivating our trace architecture, we present the event pipeline and its context management followed by a description of file and buffer handling. In a second time we introduce the low level format used to encode events within the trace.

9.4.1 Topology Management

In order to simplify topology management while allowing the storage of events in separated files, the MPC Trace library relies on a hierarchical trace format which follows the effective topology. This approach simplifies identifier management as they are all local during instrumentation and unified on-the-fly upon trace reading by the MPC Trace reader.

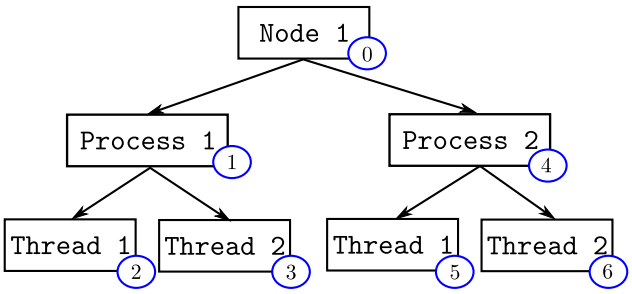


Figure 9.8: *Example of hierarchical trace with identifiers.*

As depicted in figure 9.8, unique identifiers are computed through a Depth First Search (DFS) in the topology tree, numbering streams by adding parent prefix to local identifier. When dealing with threads, events can be multiplexed in the same file in order to limit the number of files, such files are handled by adding the number of multiplexed threads to the identifier counter. This process avoids complete trace rewriting while providing support for several topologies (including MPC). It also allows partial trace processing while preserving unique identifiers. At instrumentation time this hierarchy is maintained in a tree of `Trace_Modules` which are stored as Thread Local Storage (TLS) values.

9.4.2 Event Description

In order to simplify the data-path, trace events are represented in memory as a single hierarchical data-structure called the `Generic_Event`. Each event is located in both space and time and features a type, a sub-type and a payload which consists in several `uint64_t` values. As presented in figure 9.1, this simplifies event handling as they follow a single path until being stored in the trace. Similarly, upon reading, events can be submitted to a single handler. Moreover, on debug buffers side, this generic representation allows the storage of debug events in simple arrays. Figure 9.9 illustrate our event representation for common events with a set of common attributes and event specific fields.

9.4.3 File Descriptor Handling

As file-descriptors count is limited (generally at 1024 file-handlers), it is crucial to limit the number of opened files when manipulating large traces. Besides, as we design our trace format to run in a highly multi-threaded environment, we have to make sure files are managed in a thread-safe fashion, without impacting performances. This led to the development of our `File_Handler` which relies on a file abstraction called the `TL_File` in purpose of hiding file management complexity from readers and writers.

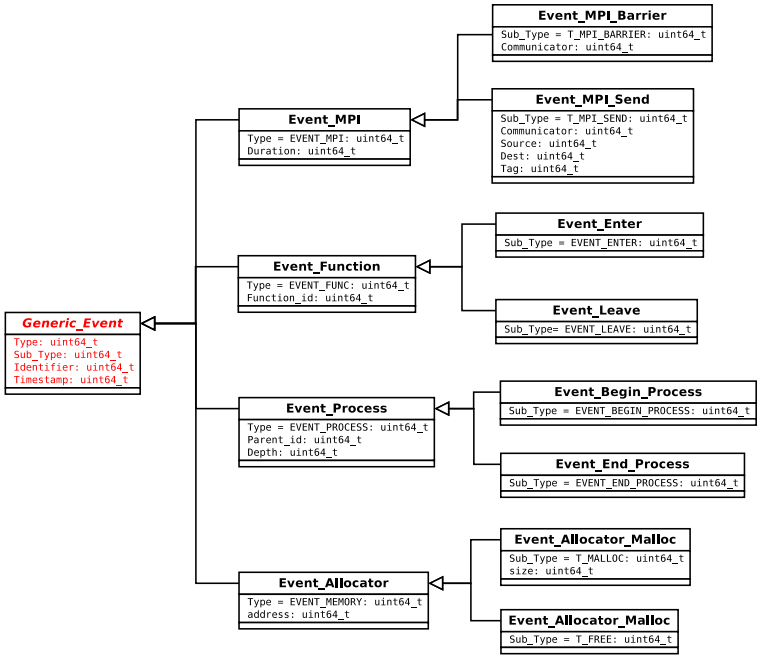


Figure 9.9: *Illustration of our hierarchical event representation.*

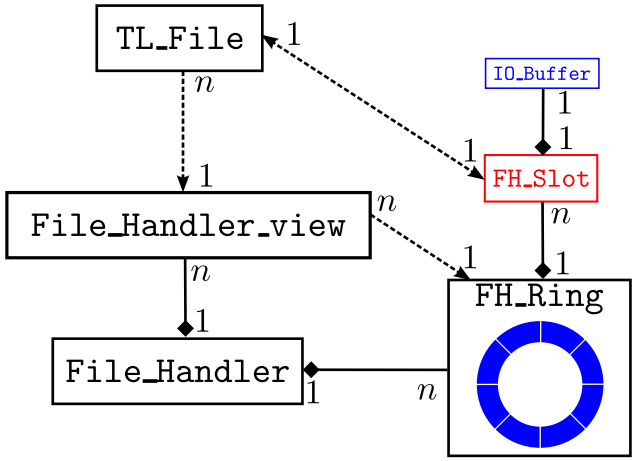


Figure 9.10: *Architecture of our thread-safe* `File_Handler`.

Figure 9.10 presents our File_Handler, it is build around `FH_Slots`, embedding IO_Buffers which have an opened file handler either in reading or writing mode. Theses slots are gathered in a ring, located at processor level (when running over MPC). Rings which are gathered in the main File Handler structure. In order to balance the load among threads, a `File_Handler_view` structure is stored as a thread local storage (TLS) value upon thread creation in order to point towards a ring. Thus, TL_Files rely on the thread-local view to query an FH_Slot within the ring, slots which can be in three states:

- **`IN_USE`**: a TL_File is currently reading or writing through this slot.
- **`RELAXED`**: this slot is held by a TL_File but not in use.
- **`UNINITIALISED`**: This slot is not used.

Consequently, when a TL_File is about to read or write data it has to acquire a slot (through the view) by querying the ring in search of either an (1)`UNINITIALISED` slot or a (2)`RELAXED` one. In the first case the file is simply opened and the call returns. In the second case, current file offset is stored in the TL_File and the slot is stolen, flushing the buffer if needed then closing previous file before opening a new one. If no slots are available at a given moment, the waiting threads sweeps the ring until finding an empty slot, process which aims at limiting contention. This approach has the advantage of allowing thread-safe file management while limiting the total buffer size as buffers are stored in slots and limited in total number independently from the number of threads.

9.4.4 Debug Buffers

In order to use a trace for debugging purpose, a program must be able to dump its n last events even if signaled. Classical trace writing through output buffers does not guaranty that all events were flushed to disk. Therefore, in case of crash, traces can be truncated, failing at describing the final program state. As hierarchical events have a fixed width, they can be stored for each process in a ring buffer located in a shared memory segment setup by the launcher (see figure 9.1 and 9.11). In this configuration, if a process crashes, shared buffers' content and either the return value or killing signal of the instrumented program can be stored by the launcher in a valid trace.

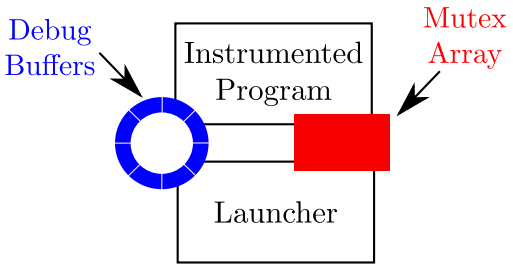


Figure 9.11: *Instrumentation coupling through shared memory segments.*

Besides, as presented in figure 9.11, each lock final state can be retrieved using a mechanism similar to debug buffers. A mutex state is a tuple $(\mathrm{Address}, \mathrm{Holder}, \mathrm{State})$ with $\mathrm{Address}$ being the mutex address, Holder currently or previously holding stream identifier and State, resource status. The launcher creates a shared memory segment of mutex states with a footprint of one megabyte for 32768 mutexes. At runtime, mutex states are updated by hooks on mutex related calls. A hash table associates each mutex address with a state residing in the shared memory segment. If there is no state associated with the requested address, a new segment is booked, within the limit of available states. As these modifications are done either after taking the lock or before its release, it is the instrumented mutex which ensures its own state update atomicity. Thanks to these mechanisms, the n last operation and mutexes' final status can be stored in a valid trace, even in case of crash or interruption.

9.4.5 Symbol Extraction

In order to associate functions calls (particularly compiler instrumented function) with code locus, symbol information have to be extracted from libraries linked to the instrumented exe-

cutable. To do so our approach relies on an analysis of the ELF[2] binary format [C+95] (which is common to most UNIX systems), combined with the DWARF [C+10] debug information which are appended to the executable by the compiler (flag -g) when compiled in debug mode.

Library Extraction

In order to extract libraries which are linked to a given program, our instrumentation library relies on a linux facility provided by */proc/self/maps*. Using this file a program can perform an introspection to list its own dependencies in order to extract their symbols.

```
$cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:05 2097188            /bin/cat
0060a000-0060b000 r--p 0000a000 08:05 2097188            /bin/cat
0060b000-0060c000 rw-p 0000b000 08:05 2097188            /bin/cat
01649000-0166a000 rw-p 00000000 00:00 0                  [heap]
7fcc051c6000-7fcc058b2000 r--p 00000000 08:05 2759235    /usr/lib/locale/locale-archive
7fcc058b2000-7fcc05a67000 r-xp 00000000 08:05 4329379    /lib/x86_64-linux-gnu/libc-2.15.so
[..]
7fff1a406000-7fff1a427000 rw-p 00000000 00:00 0          [stack]
7fff1a517000-7fff1a518000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Figure 9.12: *Output (truncated) of /proc/self/map for the cat command*

Moreover, this command also provides the base mapping address of each library (leftmost value in figure 9.12), loaded by the loader at various offsets for security reasons. This value is important as when stored in a shared library symbols are *position independent* this means that calls to these functions are done through the *Procedure Linkage Table* (PLT) which 'redirects' them to the actual function address as resolved by the linker. Therefore, when extracting symbols, as detailed in next section, this base address is required to derive the actual address.

Symbol Table Reading

The symbol table is processed using the libelf [Kos10] library which provides useful primitives to explore the ELF executable layout. As presented in figure 9.13, an ELF executable is organised in different sections with different names, types, attributes, size, member data and file offset.

Among the main sections are:

- **.text:** which gathers program instructions. As this section has the *load* attribute at '1', it is loaded in memory upon program execution.
- **.debug:** this section is present if the program has been compiled with the '-g' option. It contains DWARF informations which will be presented in the following section.
- **.symtab:** it is the section we are interested in this section, it lists all the symbols contained in the executable.

[2] Executable and Linkable Format

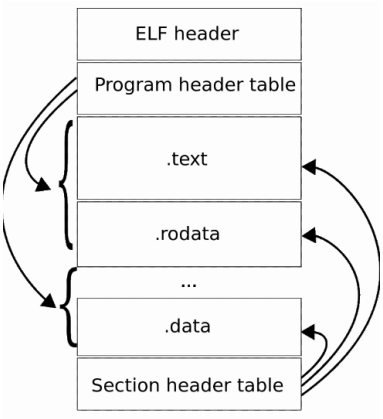


Figure 9.13: *ELF executable layout.*

It is the type of section which determines the semantic of the data it contains. Thus, the symbol table can be locate by looking up its identifier (SHT_SYMTAB) among the various sections headers. As shown in figure 9.13, the file starts with a fixed header which regroups global information such as the target architecture. Then, the program header describes how to execute the code (entry point). Eventually, several sections headers describe various sections which are located at a given offset in the executable. As mentioned before, informations we are interested in are located in the symbol table (.symtab) section, the debug information section (.debug) and the program (.text) section for the assembler code itself. Program header also contains important values such as the $\mathrm{p\_vaddr}$ address which gives the offset of the first byte where the .text section will be loaded in virtual memory and also the $\mathrm{p\_offset}$ which gives this offset in the executable file itself.

Symbol table contains the matching between symbol names and their address (either relative or absolute). In order to calculate their effective address, when mapped dynamically, we read the */proc/self/maps* file and process executable segments (r-xp in figure 9.12) in order to extract their mapping addresses. Thanks to the ELF format, it is then possible to compute the effective address of any symbol either in the memory or in the executable:

- **Absolute address:** this is the address where the function is actually loaded in memory. It is computed using the relative address in the executable (or .so file) from which we subtract the $\mathrm{p\_vaddr}$ address which is located in the program header in order to have a zero based address space. Then, we simply add the base address of the memory segment where this library is loaded to retrieve the actual address. This process is summed up in equation 9.1.

$$\mathrm{Addr}_{\mathrm{absolute}} = \mathrm{Addr}_{\mathrm{relative}} - \mathrm{p\_vaddr} + \mathrm{Addr}_{\mathrm{library-base}} \tag{9.1}$$

- **Executable offset:** the executable image loaded in memory is the exact copy of a section of the executable. It is then possible to compute the offset of a function in the executable in order to compare the absolute address computation by checking whether they contain the same instructions. As before we get back to a zero based addressing but instead of adding the library base address, we now add the $\mathrm{p\_offset}$ value which is the base address of the .text section. Computation depicted in equation 9.2.

$$\mathrm{Offset}_{executable} = \mathrm{Addr}_{relative} - \mathrm{p\_vaddr} + \mathrm{p\_offset} \tag{9.2}$$

Thanks to these calculations symbol resolution, including dynamic ones is performed when instrumenting programs. This process is presented more practically in section 9.4.5 where we also introduce our approach for function identifier homogenisation.

Debug Information Processing

Once symbols are resolved to their address, they still have to be projected on the source code. To do so, we rely on the DWARF debugging information [C+10] which names comes from the analogy with ELF. These information are appended by the compiler when the binary is compiled in debug mode. In order to parse the DWARF format, we rely on the libdwarf library for code locus information extraction.

DWARF relies on a basic structure to store debugging information: the Debugging Information Entry (DIE). These DIEs are grouped in Compilation Units (CU) which match a given program portion (division is done at source-file level). An ELF executable embeds DWARF information which are gathered in a tree fashion where multiple CUs contain several DIEs. There are several types of DIEs, but for what we are interested in, we only retain the DW_TAG_subprogram which describes functions. During instrumentation, CUs are walked through recursively in search for DIEs which describe functions (DW_TAG_subprogram). Then for each of those entries, we retrieve the declaration source file (DW_AT_decl_file) and the source declaration line (DW_AT_decl_line). Consequently, if programs were compiled with debug enabled, we can display the symbol location in the "source.c:line" form.

Distributed Symbol Resolution

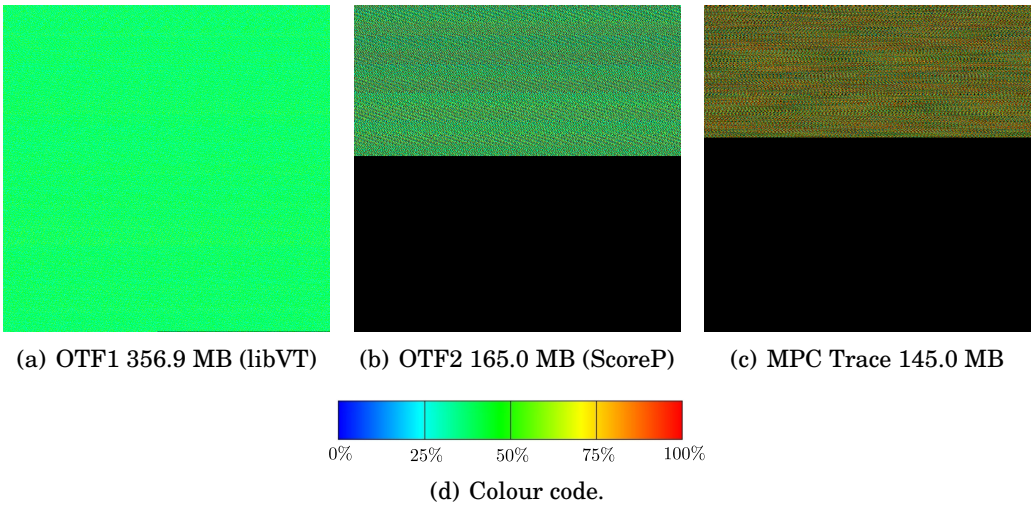
One problematic when resolving symbols is to associate compact identifier which each function calls in order not to write (generally large) addresses to describe function calls. Moreover, when running on several processes (and possibly several nodes), absolute addresses of dynamically loaded functions can vary as libraries are not loaded at a predefined address. Consequently, this unique identifier is also a way of unifying function descriptions. In order to retrieve global identifiers, we sort libraries using a sum of control of their executable, assuming they are identical on every nodes thanks to the shared file-system. Then we process each library in order, loading symbols which order is identical at executable level, allowing the use of a simple counter for identifier generation. Apart from the Multiple Program Multiple Data (MPMD) mode where different programs are running, it is then possible to generate a list of tuples which provide instrumentation with the following context:

$$(\mathrm{Function\ Name}, \mathrm{Address}, \mathrm{Identifier})$$

These informations are gathered in a hash table to allow dynamic symbol resolution (address $\rightarrow$ identifier), therefore, storing only global identifiers in the trace. Moreover, thanks to the DWARF debug data, each of these function can be located in terms of source code. Eventually, all those information are stored in the trace in order to be retrieved at reading time in order to build a look-up table (as identifiers are contiguous), allowing identifier matching in a constant time during the distributed analysis.

9.4.6 Compression

This section develops the trace compression method used by the MPC Trace format. After detailing methods, used by most common performance trace formats which are OTF1 and OTF2, we introduce our method and its implementation relying on a binary truncation method (similar to OTF2) but which yields slightly smaller trace thanks to a sub-byte value width storage.



(a) OTF1 356.9 MB (libVT) (b) OTF2 165.0 MB (ScoreP) (c) MPC Trace 145.0 MB

(d) Colour code.

Figure 9.14: *Bit-density when tracing a simple program (MPI only, instrumenting the same events) with OTF1, OTF2 and MPC Trace.*

In order to reduce traces volume, trace formats often involve compression or try to reduce events verbosity by relying on compact event representations. For example OTF1 [KBB+06] uses an ASCII-based hexadecimal representation which ignores leading zeroes, ensuring trace portability and slightly reducing trace size when compared to raw data. Other formats adopted a simple textual representation [dOSdKM10], when others [WM04, CGL08] among which OTF1's successor's OTF2 [EWG+11, aMBB+12] rely on a binary representation with the extended possibility of being compressed with state of the art algorithms such as zlib. Figure 9.14 illustrates this storage layout difference between state of the art trace formats OTF1 and OTF2. OTF1 relies on an ASCII-based representation which ignores leading zeroes, thus, for example if the value 65535 has to be stored, it will be converted to hexadecimal over 64 bits, yielding *0000'0000'0000'FFFF* but the leading zeroes can be clearly ignored in order to save space to get *FFFF*, value which is then stored in ASCII with a cost of eight bits per symbol *0100'0110'0100'0110'0100'0110'0100'0110* (in ASCII, 'F'=0100'0110 in one Byte). Moreover, as it can be seen in figure 9.14(a), symbol dynamic is very reduced as it is bounded to the ASCII 0-9 (encoded 48-57) and A-F (encoded 65-70), yielding this very regular bit distribution as those values are very close. If we try to compare the two compression approaches of OTF1 and OTF2, we can derive the entry size in function of the stored value and determine if this approach is efficient compared to directly storing the raw value. As truncation compression depends on value *width* when written in binary, we first need to derive analytically the offset of the most significant bit in a binary value:

Consider $X \in]2^{n-1}, 2^n]$, $1 < n$, $1 < X$

$$2^{n-1} < X \leq 2^n \tag{9.3}$$
$$e^{\ln(2)(n-1)} < X \leq e^{\ln(2)(n)}$$
$$\ln(2)(n-1) < \ln(X) \leq \ln(2)n$$
$$n-1 < \frac{\ln(X)}{\ln(2)} \leq n$$
$$n-1 < \ln_2(X) \leq n \tag{9.4}$$

If we consider a value X such as $X \in]2^{n-1}, 2^n]$, it has by definition a width of n. It is then possible to write this condition (see equation 9.3) in order to derive a way of bounding X in terms of its width in equation 9.4. Yielding $l_2(X)$ the width when stored in base 2 of a value X:

$$l_2(X) = \lceil \ln_2(X) \rceil \tag{9.5}$$

Process which can be repeated in base 16 to derive a similar equation:

$$l_{16}(X) = \lceil \ln_{16}(X) \rceil \tag{9.6}$$

Base 10	l_{10}	Base 2	l_2	Base 16	l_{16}
1293	4	101'0000'1101	11	50D	3
32496	5	111'1110'1111'0000	15	7EF0	4
344216597562	12	101'0000'0010'0100'1110'1000'0111'0100'0011'1010	39	5024E8743A	10

Figure 9.15: *Illustration of width calculation for some values in base 10, 16 and 2.*

Figure 9.15 illustrates width calculation for various values, depicting the variation in symbol count depending on the base. Hexadecimal seems to rely on less symbols than decimal which itself has less symbols than binary, assuming a constant symbol size. However, symbol size is not constant and if '0' and '1' have a size of one by definition of the architecture, numbers and hexadecimal symbols have a width of eight binary symbols as they are stored in ASCII. Moreover, the word size on our architecture is 8 bits or one byte, this means than raw values can only be truncated in blocks of eight bits. Therefore we have to adjust formulas 9.5 and 9.6 to take those underlying storage requirement into account:

$$l_{2-\text{Bytes}}(X) = 8\lceil \frac{\ln_2(X)}{8} \rceil \tag{9.7}$$
$$l_{16-\text{ASCII}}(X) = 8\lceil \ln_{16}(X) \rceil \tag{9.8}$$

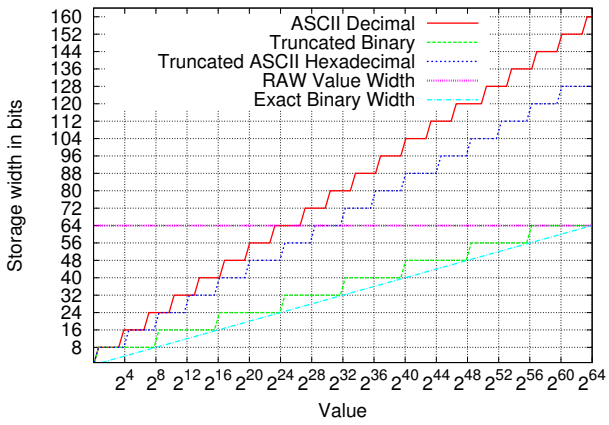
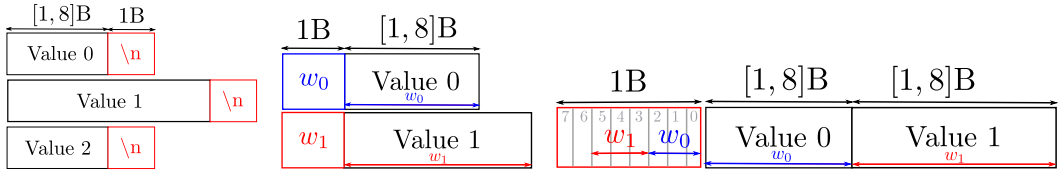


Figure 9.16: *Comparison of various storage methods for 64 bits unsigned integers, neglecting context information such as with storage or separators.*

As outlined in figure 9.16, if we consider gains obtained from the truncation itself it can already become inefficient. If we consider the ASCII-Hexadecimal truncation it becomes sub-optimal at 2^{32}, value which despite relatively large is common in performance traces as it can be reached in a time $T = \frac{2^{32}}{2\times10^9} = 2.15\ seconds$ with the TSC (see section 6.2.1) on a 2 GHz machine. Similar observations can be made for the ASCII-decimal approach which is also inefficient passed a certain value. Dealing with the binary approach, it remains optimal on the whole dynamic and can compress with ratios up to 8 (8 bits over 64 for values lower than 256). However, this compression ratio does not take into account control values which are required to decode values with variable width. This leads to the comparison of these two approaches:

- **Separator approach (or byte-sized width):** which consist in separating entries with a special character (for example newline in OTF1, see Figure 9.17(a)). Note that this approach has the same cost than storing each width in a byte sized integer as the separator is also byte-sized (Figure 9.17(b)).
- **Sub-byte storage width:** as the width of a byte is in the $w \in [0, 8]$ interval, three bits are sufficient to store width. Consequently, a single byte can store two width with an extra 2 bits which can be used for other purposes (Figure 9.17(c)).



(a) Separator. (b) Byte-sized width (c) Sub-byte width

Figure 9.17: *Comparison of width storage/retrieval methods.*

From figure 9.17, we can now derive the effective truncated binary storage size for a list of N values v_i with $i \in [1, N]$. If we consider the separator (or byte-sized width) approach, we can

derive an overall storage with $W_{sep}(N)$ such as:

$$W_{sep}(N) = \sum_{i=1}^{N} l_2(v_i) + N \tag{9.9}$$

Similarly we can derive the overall truncated binary size W_{sub} for a sub-byte with storage:

$$W_{sub}(N) = \sum_{i=1}^{N} l_2(v_i) + \lceil \frac{N}{2} \rceil \tag{9.10}$$

From equations 9.9 and 9.10, it is then possible to derive the size difference between the two methods in function of the number of values stored, as the truncation uses the same width. This yield $\Delta(N) = N - \lceil \frac{N}{2} \rceil$ (in bytes):

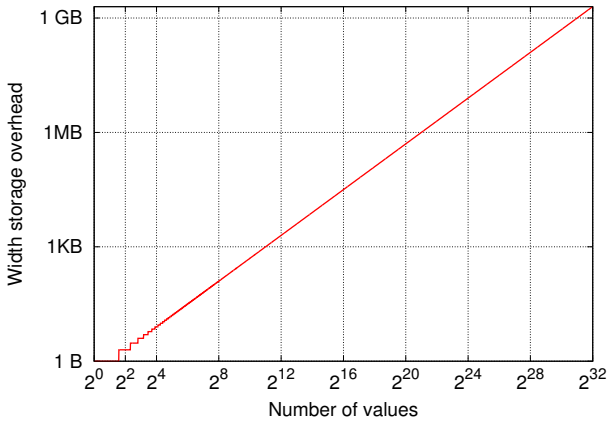


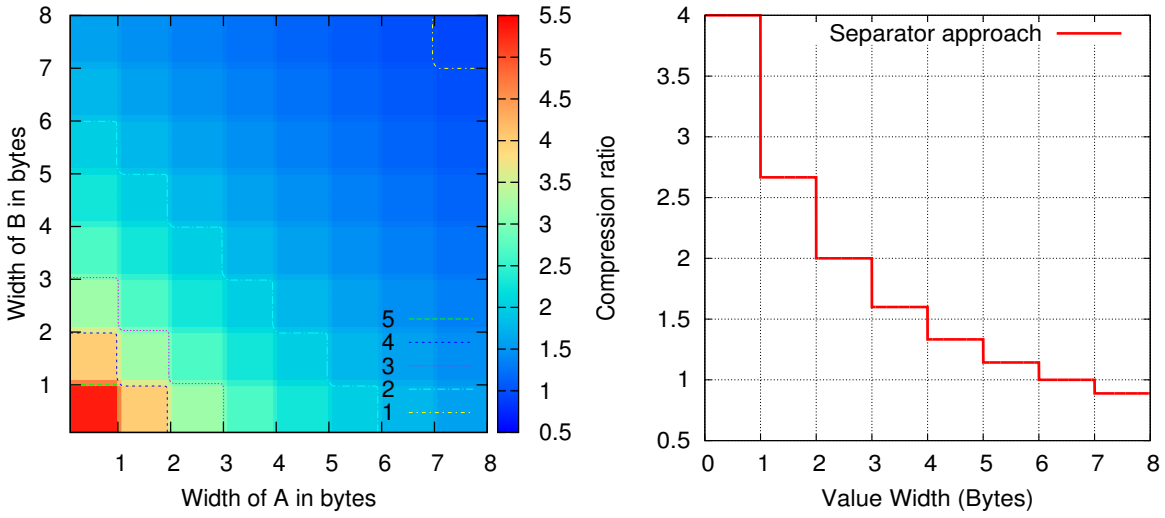
Figure 9.18: *Overall size difference for N values using either separator or sub-byte with approach for 2^{32} values (equivalent to 32 GB of trace assuming RAW 64 bit unsigned int).*

Difference which as shown in figure 9.18 tends to grow with the number of values until becoming non-negligible as sub-byte approach halves the width storage size. It is this difference which contributes to the slightly higher compression of the MPC trace format (Figure 9.14(c)) when compare to OTF2 (Figure 9.14(b)), despite using the same compression approach.

It is also possible to compare separator and sub-byte width methods by observing the overall compression ratios. If we consider the separator (or byte-sized width approach), compression ratio $R_{sep}(v)$ can be computed from the width of the truncated values v_1 in bytes as:

$$R_{sep}(v) = \frac{8}{1 + l_2(v)} \tag{9.11}$$

And the ratio R_{sub} for a sub-byte from the storage size of two truncated values v_1 and v_2 is:



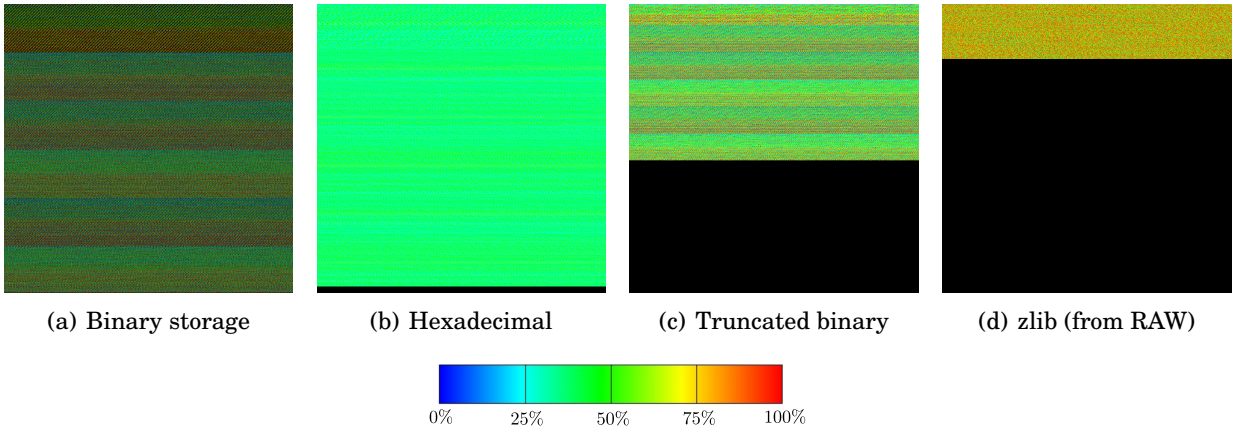
(a) Sub-byte width (Equation 9.12). (b) Separator (Equation 9.11).

Figure 9.19: *Comparison of separator and sub-byte width storage in terms of compression ratio.*

$$R_{sub}(v) = \frac{16}{1 + l_2(v_1) + l_2(v_2)} \tag{9.12}$$

As presented in figure 9.19, these two approaches have different compression ratios. The separator approach has a maximum compression rate of $R_{sep}(v) = \frac{8}{1+1} = 4$ when two bytes are used to describe a 64 bits values, dealing with the worst case, if the compressed values occupies 8 bytes, the ratio is sub-optimal with a value of 0.88. If we look at our approach in the MPC Trace format, encoding two width in the same byte leads to better ratios and lowers the worst case: highest compression is $R_{sub}(v) = \frac{16}{1+1+1} = 5,33$ when encoding two integer lower than 256. As far as the worst case is concerned, encoding two 64 bit integers yields a compression ratio of: $R_{sub}(v) = \frac{16}{1+8+8} = 0,94$, more efficient than the delimiter case.

Consequently, we have seen that when storing values which dynamic cover a wide range of the 64 bit space, it could be interesting to truncate those values. As it is the case for performance traces which gather, for example, both time-stamps (large values) and identifiers (small values), we proposed to examine various alternative to quantify their storage requirements. If we look at figure 9.20, we can see the compression of a "ramp" of values in the $[0, 1.10^8]$ range. It can be seen in figure 9.20(a) that when stored in raw, there are a lot of zeroes (coded black) which are redundant. However, as we mentioned before, using an hexadecimal-ASCII coding with a one byte separator (Figure 9.20(b)) can be inefficient for larger values. Dealing with truncated binary 9.20(c), it is still advantaging for large values but compresses more efficiently smaller values. Nonetheless, those trivial compression approaches are not comparable with state of the art compression algorithms such a zlib [DG96] (Figure 9.20(d)) which provides higher compression ratios with the drawback of an higher computational costs. Consequently, trace compression approaches does not aim at fully compressing the trace, but at finding a



(a) Binary storage (b) Hexadecimal (c) Truncated binary (d) zlib (from RAW)

(e) Colour code.

Figure 9.20: *Storage methods comparison (bit-density) for the first* 1.10^8 `uint64_t`.

trade-off between space requirements and compression costs. Balance which in our case is provided by binary truncation.

Writing Implementation

```
unsigned char buff[17];
int offset = 1;
buff[0] = 0;
buff[0] |= (type)<< 6);
buff[0] |= (write_uint64t(buff + offset, value1, &offset) << 3);
buff[0] |= write_uint64t(buff + offset, value2, &offset);
IO_Buffer_write(IO_buffer, offset, (char *) buff);
```

Figure 9.21: *Writing data using our truncation function.*

```
uint8_t write_uint64t(unsigned char *dest, uint64_t * source, int *offset){

  uint64_t width = 0;

  if (*source) {  // if source is zero bsr is undefined
    asm volatile ("bsr %1, %0":"=r" (width):"r"(*source));
    width = (width >> 3);
  } else {
    width = 0;
  }

  *((uint64_t *) dest) = *source;
  *offset += width + 1;
  return width;
}
```

Figure 9.22: *64 bit integer binary truncation implementation in the MPC Trace library.*

As presented in figure 9.22, our implementation relies on the Bit Scan Reverse (bsr) instruction which return most significant bit offset for a 64 bit integer. This values is used divided by 8 (through a 3 bit right shift) in order to retrieve the actual width in bytes. Once computed the whole value is computed and the offset is incremented by the with to indicate where to write next value. Dealing with the use of this primitive, as presented in figure 9.21 data are written in a static buffer before being written in the IO Buffer through successive calls which are 'ored' to the header. Note that the two bits left in the header are used to store a four states type which is generally sufficient to encode all sub-types of a given event as event multiplexing is limited by the trace format which separates events in multiple files.

Reading Implementation

```
static const uint64_t width_mask[] = {
  0x0llu,//0
  0xFFllu,//1
  0xFFFFllu//2
  0xFFFFFFllu,//3
  0xFFFFFFFFllu,//4
  0xFFFFFFFFFFllu,//5
  0xFFFFFFFFFFFFllu,//6
  0xFFFFFFFFFFFFFFllu,//7
  0xFFFFFFFFFFFFFFFFllu//8
};

void read_uint64_t(char *source, uint64_t * value, uint8_t width, int *read_offset) {
  *value = *((uint64_t *) source) & width_mask[width + 1];
  *read_offset += width + 1;
}

void read_header(char *source, uint8_t * v1_width, uint8_t * v2_width,
           uint64_t * type, int *read_offset) {
  *v1_width = ( ((uint8_t) * source ) >> 3 ) & 0x7;
  *v2_width = *source & 0x7;
  *type = ((uint8_t) *source) >> 6;
  (*read_offset)++;
}
```

Figure 9.23: *64 bit integer binary truncation at read time.*

On reading side, as presented in figure 9.23 the header is decoded then successive values are simply masked to retrieve actual values. Figure 9.24 presents the reading of data written in previous section. It first guarantees a sufficient size from the IO buffers assuming values are all 64 bits. Then, it decodes the header an proceeds with the reading of the two values. Eventually, if the actual width was lower than the width which has been guaranteed, the buffer is sought back of the difference in size in order to be aligned with next header.

9.5 Trace Reader

This section presents the MPC Trace reader which is a parallel trace reader which unlike most tools extends the trace format with a fully distributed analysis engine. We first introduce MPC Trace reader architecture followed by both an outline of its interface and a sample analysis tool. Then, we present some performance results, demonstrating the scalability of

```
char *buff = NULL;
int count = IO_Buffer_read(IO_buffer, (char **) &buff, 17);
uint8_t v1_width = 0;
uint8_t v2_width = 0;
uint8_t type = 0;
int read_offset = 0;
uint64_t v1 = 0;
uint64_t v2 = 0;

read_header(buff, &v1_width, &v2_width, &type , &read_offset);
read_uint64_t(buff + read_offset, &v1, v1_width, &read_offset);
read_uint64_t(buff + read_offset, &v2 , v2_width, &read_offset);
buffer_seek_back(IO_buffer, count - read_offset);
```

Figure 9.24: *Reading data using our truncation function.*

this approach on thousands of cores. Eventually, we list the limitations of this approach which led to the on-line method, we develop in chapter 10.

9.5.1 Trace Reader Architecture

The trace reader abstracts the analysis by handling transparently: (1) distributed event reading, (2) context information management and dispatch and (3) analysis parallelism. Therefore, this trace reader allows the simple design of scalable tool associated with our trace format while limiting implementation complexity as most common operations are managed by the trace reader.

Distributed event reading is performed using a simple producer consumer-model. Indeed, when processing the trace in depth-first-search, files are identified as separated tasks which can be sent to different processes which are in charge of reading them. There are options to regroup files from the same process in the same analysis process in order to allow local analysis (particularly deadlock detection). But, more generally files can be dispatched indifferently. *Context information* are gathered by the root process which scans the trace in order to list trace files while generating unique identifiers. Identifier look-up tables regroups individual stream description including: Stream type (Node, Process, Thread), start and end-times, hostname and parent identifiers (-1 if trace root). Dealing with function descriptions they are stored as code locus including source library and source code level location (file:line). As far as *parallelism* is concerned, the analysis engine relies on a mixed pthread plus MPI task engine which processes a set of tasks which are statically assigned when opening the trace. If the number of files to process is larger than the number of processing units, it can cause temporal scattering as event order cannot be guaranteed, files being read successively. Behaviour which prevents analysis depending on multiple files which processing has to be temporally correlated. However, we have no such analysis as we managed to keep a global state when operating the reductions even temporal ones by using state matrices which can put up with unordered file reading (at file level, not event level which are ordered by construction in the trace file).

9.5.2 Trace Reader Interface

Figure 9.25 presents the MPC trace reader interface, as aforementioned it handles both parallel trace reading and meta-data through a compact interface which as illustrated in next section can be used to build distributed trace processing tools in only a few lines.

Function	Description
MPC_Trace_reader_init	Initialise distributed trace reading on a given trace.
MPC_Trace_reader_release	Release the trace reading interface.
MPC_Trace_handler_attach	Attach an handler to a given event type.
MPC_Trace_read_definitions	Process trace hierarchy in order to load meta-data.
MPC_Trace_reader_wait	Wait until the trace has been fully processed.
MPC_Trace_id_infos	Retrieves the description of a given identifier.
MPC_Trace_id2rank	Get the MPI rank of a given identifier.
MPC_Trace_func_infos	Retrieves a function description from its identifier.
MPC_Trace_get_begin	Returns the time-stamp upon application start (in ticks).
MPC_Trace_get_end	Returns the time-stamp upon application end (in ticks).
MPC_Trace_get_duration	Returns application duration (in ticks).
MPC_trace_type	Returns trace type (MPI, Pthread, MPC).

Figure 9.25: *MPC Trace reader interface.*

9.5.3 Sample Tool

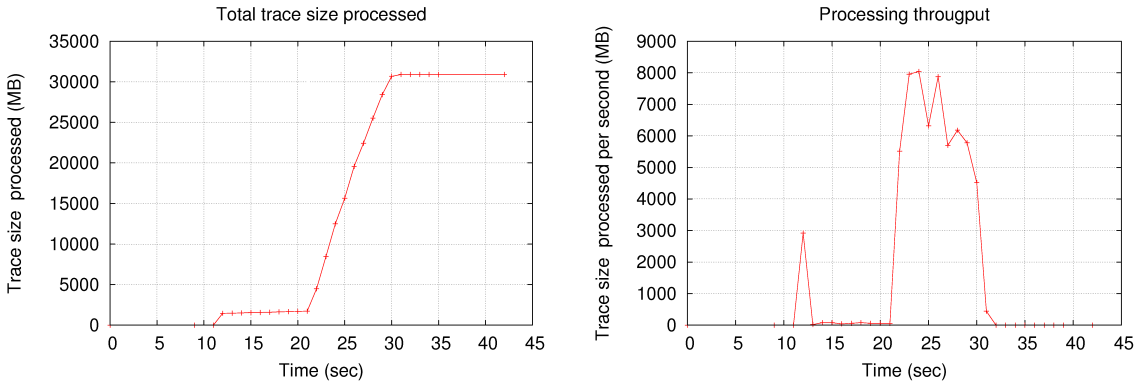
```
void comm_handler( struct Gen_Event_t *event, void *dummy) { /* Process Events */ }

int main( int argc, char **argv ) {
  int pr;
  MPI_Init_thread( &argc, &argv, MPI_THREAD_MULTIPLE, &pr);
  MPC_Trace_reader_init("./trace/", 100, 1024*1024*10, 1, NULL);
  MPC_Trace_handler_attach( EVENT_MPI, comm_handler, NULL );
  MPC_Trace_read_events();
  MPC_Trace_reader_wait();
  MPC_Trace_reader_release();
  MPI_Finalize();
}
```

Figure 9.26: *Example of tool relying on the trace reader interface.*

Figure 9.26 presents a minimal analysis tool which relies on the MPC Trace reader interface. We first initialise the MPI environment before opening the trace located in *./trace/* with a maximum of 100 file descriptors and IO Buffers of 10 MB with one thread per MPI process and no handler (NULL) associated with trace reading completion. Then, we register the `comm_handler` event to MPI events with no extra arguments. Eventually, we enter distributed event reading loop and wait for its termination before releasing the environment.

9.5.4 Performance

Figure 9.27 presents MPC Trace reader performance when processing a trace of lbm (see section 12.1) on the Curie supercomputer in terms of both total trace size processed (Figure 9.27(a)) and processing throughput (Figure 9.27(b)). Measurement were done over applications wall-times, including initialisation and trace processing costs. It can be seen that this 30 GB trace is processed in 43 seconds, including report rendering. Dealing with the throughput, we can observe three main phases: (1) (0-8 seconds) trace processing by the root process in order to list files while accumulating and broadcasting topological information, (2) (8-30 seconds) distributed trace processing with a slow start until all files are opened, delivering the maximum processing throughput (≈8 GB/second) and eventually, (3) (30-43 seconds) performance data reduction and trace report rendering by the root process.



(a) Total trace size processed.

(b) Trace Procession throughput.

Figure 9.27: *MPC Trace reader performance for a 128 core run of lbm (see section 12.1) processed on 128 cores on the Curie supercomputer.*

9.6 Limitation

One of the main limitation of the trace analysis framework we introduced in this chapter is the large number of files its relies on. Indeed, by avoiding event multiplexing in the same file, we can possibly create up to three files per core, approach which is definitively not scalable on more than a machine fraction, because of IO limitations. Despite preserving analysis parallelism, avoiding multiplexing, this approach is therefore not sustainable to fulfil our performance requirements which aim at building a scalable tool. Another aspect is trace format complexity as new events have to be defined into the trace format, despite being represented by generic events during the whole instrumentation chain. This was done this way in order to write the minimum amount of data instead of always compressing the whole structure, but with the passing of time, this approach was error prone and discouraged trace format updates. As a consequence, latter implementation have to rely on a simple event representation scheme with an agnostic transport layer. Moreover, extension has to be part of the design as there are always new events to add. Consequently, although providing useful features, our trace-based approach suffers from major limitations which prevent it to be used a machine scale because of IOs, motivating the transition towards our on-line tool presented in Chapter 10.

9.7 Summary

This chapter introduced the first version of our tool called the MPC Trace library. It aimed at instrumenting both MPC and MPI programs in a scalable manner while providing tools with a generic interface. We firstly presented a new trace format which aimed at overcoming some limitations of OTF1[3] while supporting MPC's extended topology. We also described a method allowing trace-based debugging of faulty programs thanks to a shared memory segment flushed upon program interruption. Subsequently, we presented a compact distributed trace processing interface which can be used to provide analysis with mixed parallelism through handlers' control flow (MPI + Pthread). Eventually, we discussed of the limitation of this approach, outlining the unacceptable number of files it creates and a lack of

[3] Limitations which were mostly resolved in OTF2.

adaptability because of trace format adherence. Observations which motivated a full rewrite of our tool in the light of this first experiment, leading to the development of Chapter 10 which deals with the Multi-Application on-Line Profiling (MALP) tool.

CHAPTER 10

Online Trace Analysis

This chapter presents the second version of our profiling tool. This tool called *MALP* for Multi-Application Online Profiling has been designed to avoid the IO bottleneck by relying on an alternative coupling method. We start by justifying the shift from a trace based to an on-line coupling method. Then, after introducing our multi-application coupling mechanism based on MPI in MPMD, we present a data-flow engine inspired from blackboard systems which purpose is to perform a parallel event reduction. This work has been published in an article [BPJ13] which content is partially reproduced in this chapter.

10.1 Shifting to On-line Trace Analysis

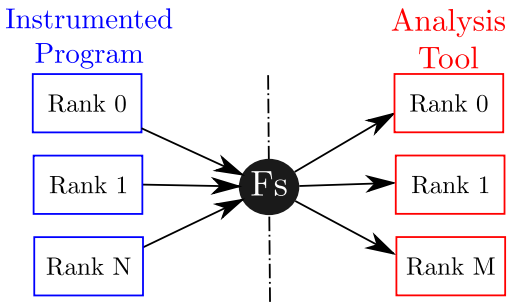


Figure 10.1: *Overview of the trace-based instrumentation coupling.*

As developed in chapter 9, our previous approach was subject to IO contention and therefore unable to reach larger scales without suffering from unacceptable overhead. A common way of solving this issue is to rely on parallel IO libraries, approach adopted for example by the ScoreP [aMBB+12] framework which relies on SionLIB [FWP09]. This method takes advantage of data multiplexing in order to reduce file-system contention by limiting the number of files. In this purpose, as presented in Figure 10.1, trace data are firstly spatially reduced (generally through the MPI layer) before being written in a limited number of files. Upon reading, the same process in done in reverse, files are opened by a subset of processes which are in charge of redistributing data to every processes. This multiplexing supposes that the trace is auto-coherent, allowing its reading in a space independent way — necessarily leading to a trace format definition. Moreover, traces can be very large and generally grow linearly with

the number of processes — posing the question of disk usage for long runs[1] at larger scales. From a more general point of view, file-system is becoming a critical service at petaflopic-scale: file system providing a global view, there is necessarily a bottleneck at meta-data servers level when processing requests from thousands of processes. Observation which motivated the shift to another type of coupling which would not rely on file-system.

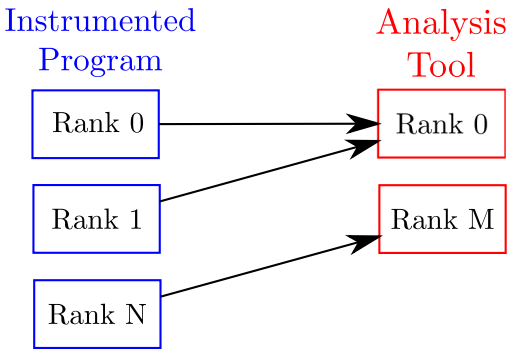


Figure 10.2: *Overview of the network-based instrumentation coupling.*

Figure 10.2 presents our on-line coupling approach where instead of transiting through file-system, trace data are directly sent to the analyser through the high-speed network. By not using file-system, temporary trace data are not stored at all and do not have to go to a slower medium. Moreover, as performance measurements are directly sent to the analyser, they avoid going back and forth between file-system's servers, halving the overall network traffic. Streaming performance data also opens new parallelism opportunities by allowing pipe-lined and concurrent analyses which were not practical with traces. However, in contrast with trace-based approach, this method does not allow iterative measurements processing as their storage is limited to the amount of memory available in processing nodes. As this process (just as profiling) relies on an immediate valuing of instrumentation data, information which are not explicitly analysed (reduced) during the measurement phase are lost. Nonetheless, analysis close to post-mortem ones are still possible, analysing processes being able to communicate during the reduction process — opening opportunities for distributed analysis. 11.

On-line coupling has been retained for instrumentation–analysis in the MALP tools as it alleviates file-system limitations by not using it at all. Moreover, as we will show, it provides opportunities to simplify the trace format if the coupling exposes a surjective propriety. This avoids temporary storage of redundant trace data by processing them on-the-fly. It also allows the analysis of long running applications which would exhaust storage capabilities if instrumented with traces. Besides, as presented in next section, on-line coupling permits multi-instrumentation which is a particularity of MALP. This approach is therefore more suitable to prototype a machine wide server which would provide profiling as a service — matching more closely our *continuous profiling* requirements.

[1] Some programs can run for several day and sometimes even weeks.

10.2 Coupling Multiple Applications

This section introduces the coupling method which is used by MALP in order to efficiently achieve runtime coupling. This mechanism is used to realise the coupling of our instrumentation chain, with the originality of being able to couple multiple applications. After introducing the architecture of our coupling mechanism, we detail its programming interface and provide an example of N to one coupling, similar to what is done when instrumenting multiple programs. Eventually, we present some performance results related to this coupling mechanism.

As detailed in the rest of this section, in order to implement an on-line coupling method, three conditions must be satisfied :

- **Transparent cohabitation:** two programs have to be able to run concurrently, in accordance with machine's scheduler.
- **Mapping:** groups of processes must map to each other.
- **Communication:** each process has to efficiently communicate using a persistent asynchronous data stream.

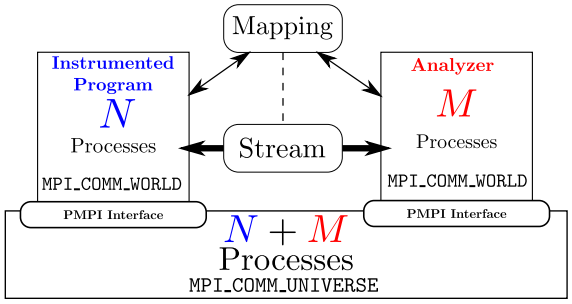


Figure 10.3: *Overview of the runtime coupling mechanism.*

As presented in Figure 10.3, these three requirements are satisfied in our implementation using MPI in MPMD mode. In this purpose, we provide transparent cohabitation with *MPI virtualization*, mapping with the *VMPI_Map* primitive and asynchronous communication with *VMPI_Streams*.

10.2.1 Transparent Cohabitation (Virtualization)

Virtualization is achieved using a simple mechanism similar to the one implemented in P^{N}MPI's virtual module [SdS07]. It consists of replacing every references to `MPI_COMM_WORLD` by a reference to a sub-communicator. This is done by intercepting every MPI calls through the PMPI interface. Originally implemented over P^{N}MPI, we had to rewrite our own virtualization outside of P^{N}MPI to provide an integrated library which can be preloaded on MPI programs without code modification, recompilation or binary patch. Moreover, providing module's interface to the host application was not convenient as our library was divided in two, separating streams and virtualization. Therefore, we decided to implement our library in a single package which can be easily linked or preloaded in order to virtualize a program. To do so, we wrote in C a MPI wrapper generator, very similar features as P^{N}MPI's python one,

with some extra options such as conditionals. Using this wrapper, we are able to generate a complete virtualization interface directly compiled into the VMPI library. Thanks to its extended interface, this wrapper was also used to generate the PMPI interface used by our instrumentation library. When running virtualized, each program runs transparently, in its own `MPI_COMM_WORLD` whereas the real `MPI_COMM_WORLD` is still available to perform inter-application communications as `MPI_COMM_UNIVERSE` (see Figure 10.3).

Wrapper Generator

The wrapper generator is used to manipulate the whole MPI interface at once, simplifying tool development. Our approach is close to P^{N}MPI's python wrapper [SdS07], although we extended its interface to match our requirements. We developed our wrapper in C and used it to generate several MPI related files at both MPI virtualization and instrumentation level.

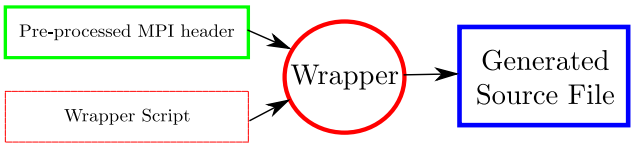


Figure 10.4: *Presentation of wrapper work-flow.*

As depicted in Figure 10.4, our wrapper relies on a pre-processed version of the MPI header as retrieve from the compilation of an empty main (with *mpicc -E ./main.c*). Data-stream from which all MPI related calls are extracted. Then, we proceed by loading the wrapper script which represents generated file skeleton. It contains arbitrary segments of code mixed with generated regions delimited by two html like tags: "<MALPW>" and "< \MALPW>". Each of these section is repeated for each MPI calls with specific command which can be used to retrieve call related information to generate the target code. Moreover, each generated segment can be preceded with either a white-list or a blacklist in purpose of targeting or ignoring specific functions, with for example, the following syntax in the opening tag: "<MALPW !MPI_Init,!MPI_Init_thread,!MPI_Finalize>" to blacklist `MPI_init`, `MPI_init_thread` and `MPI_finalize` calls.

Command	Effect
MALPW_WRAP(T, N, M)	Apply macro M to argument of type T and name N.
MALPW_PROTOTYPE()	Print function prototype
MALPW_PRINT(T)	Add text T to output file
MALPW_CALL(P)	Generate a call to MPI function prefixed with P (usually P='P'), return value is stored in 'ret' which is defined with the right type.
MALPW_WNAMECASE_UP(M)	Print current function name in upper case, processed by macro M
MALPW_WNAME(M)	Print current function name, processed by macro M
MALPW_HAS_ARG_B(T, N)	Begin a conditional section which is executed only if there is argument of type T and name N is present
MALPW_HAS_ARG_E()	End a conditional section

Figure 10.5: *List of MALP's MPI wrapper commands.*

```
#define MACRO_MPI_Comm(_c)    if (_c==MPI_COMM_WORLD)\
                                  _c=VMPI_Get_partition_comm();\
                              else if( _c==MPI_COMM_UNIVERSE)\
                                  _c=MPI_COMM_WORLD;
/* Lets wrap all MPI functions except those we already defined */
<MALPW !MPI_Init,!MPI_Init_thread,!MPI_Finalize>
MALPW_PROTOTYPE();
{
  MALPW_WRAP( MPI_Comm , * , MACRO_MPI_Comm );

  MALPW_CALL(P);
  return ret;
}
</MALPW>
```

Figure 10.6: *Wrapper script performing MPI virtualization.*

Figure 10.5 lists commands which are used to generate wrappers and context files. For example, Figure 10.6 presents the script file which is used to generate a virtualization wrapper for every MPI functions. To do so, standard code is mixed with generated regions generating MPI call dependent lines. The "MALPW_WRAP" call applies the macro "MACRO_MPI_Comm" to every (*) arguments of type "MPI_Comm". If we look at the "MACRO_MPI_Comm" itself, it just replaces references to `MPI_COMM_WORLD` by the partition comm and does noting if the communicator is equal to `MPI_COMM_UNIVERSE` (implemented as a "magic" value). This automatic method reduces programming effort while limiting the risk of error. Moreover, this approach also adapts the interface to the target runtime which might not furnish every MPI calls (for example with various MPI standards). The advantage of automatic wrapping can also be attested at instrumentation level, as the generated PMPI interface contains 14446 lines generated from a script of 93 lines – saving a lot of development time and avoiding redundant tasks.

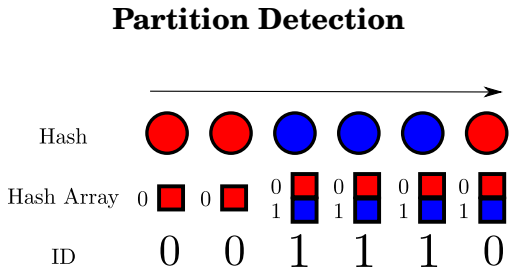


Figure 10.7: *Illustration of our partition detection algorithm.*

Partitions can be defined either from the command line or through a partition name defined by a dedicated function call. As presented in Figure 10.7, partition detection is done using a linear propagation of identifiers hashes (either command line or partition name). Each process either append its hash to the list before sending it (if not already present) or picks existing hash offset as its partition identifier. At this state, each process of a given group has the same identifier, and the number of partition (number of unique hashes) has been

broadcasted by the last process. Followingly, individual partition communicator are created through MPI_Comm_split with partition id as colour. Eventually, the root process (as seen from MPI_COMM_WORLD) receives all partition descriptions (name, size, root, command) from each partition root (as seen from partition communicator) before broadcasting it to every processes. After this step each process has a description of each partition.

VMPI Partition Management Interface

As presented in Figure 10.8, VMPI provides a compact interface to access and query partition list. VMPI can be used in two ways, either with a built-in virtualization interface which can be preloaded to transparently separated applications in function of their command line. Another version, stripped from this interface can be coupled with another interface, for example for instrumentation purpose.

Function	Description
VMPI_Enabled	Returns one if running virtualised.
VMPI_Init	Setup VMPI environment.
VMPI_Release	Release virtualization interface.
VMPI_Get_partition_id	Get local partition identifier.
VMPI_Get_desc	Return local partition description.
VMPI_Get_partition_count	Return the number of partitions.
VMPI_Get_partition_comm	Return current partition communicator.
VMPI_Set_partition_name	Set partitions name.
VMPI_Get_desc_by_name	Returns description by partition name NULL if not found.
VMPI_Get_desc_by_id	Returns description by partition id NULL if not found.
VMPI_Display_desc	Display a partition description.
VMPI_Display_descs	Display partitions descriptions.

Figure 10.8: *VMPI Partition management interface.*

10.2.2 Mappings (VMPI_Maps)

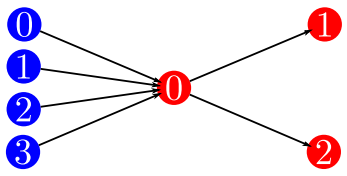
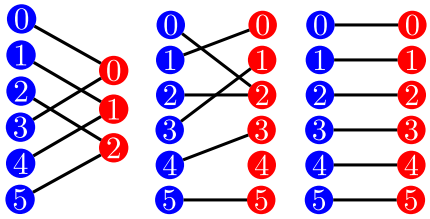


Figure 10.9: *Mapping of two partitions with a pivot.*

A basic component called VMPI_Map is provided in order to simplify process to process *mapping*. It can be used to generate a mapping between two partitions by assigning to each process a set of matching processes located in the remote partition according to a given policy. As shown in Figure 10.9, when mapping two partitions, the larger partition becomes the slave and the smaller one the master. Each process from the slave partition sends its global rank to the root of master partition (available through partition descriptions). Then, each time the master partition's root receives a rank, it picks up a local rank within its partition (including itself) according to a predefined policy, and associated local and remote ranks both-ways. In some cases, centralised mapping can be avoided when topologies can be computed locally

(for example topologies a and c of Figure 10.10). But, for specific cases, this approach allows more general mappings by providing a user-defined function which takes a source as a parameter and returns the target. Moreover, it simplifies synchronisation when handling complex topologies by providing a pivot which broadcasts the end of the mapping to every process.



(a) Round Robin (b) Random (c) Fixed

Figure 10.10: *Illustration of the three default mapping topologies.*

The three default mappings are detailed in Figure 10.10. In order to be valid for code coupling purposes, each process has to be associated with at least one other process. Partial mappings, more advanced than the trivial random one, are nonetheless possible through user-defined mapping-functions. Moreover, a VMPI_Map can be filled in an additive manner: a partition can compute its mapping to several other partitions by successively appending new entries. This feature which is particularly useful for multi-instrumentation.

VMPI_Map Interface

Function	Description
VMPI_Map_partitions	Initialise a mapping 'map' to 'target_partition' with a given 'mode'.
VMPI_Map_clear	Initialise an empty 'map' or release an existing one.

Figure 10.11: *VMPI_Map interface.*

As presented in Figure 10.11, MPI_Map interface is very compact with only two main function calls. The first one VMPI_Map_partitions can be used in a additive fashion to map multiple partitions. Whereas the second one either initialises or frees a VMPI_Map. Dealing with the VMPI_Map_mode, the three modes of Figure 10.10 are supported.

10.2.3 Communications (VMPI_Streams)

Efficient streamed-communications between partitions are provided by *VMPI_Streams* which are persistent asynchronous communication channels. They can be either multi- or uni-directional. They provide an interface and behaviour close to UNIX pipes. Writing to a stream is then non-blocking, until all asynchronous buffers are full, preserving an adaptation window between data producers and consumers. As depicted in Figure 10.12, each stream allocates buffers at both read and write endpoints to provide asynchronous streaming. Note that read endpoints have N_A (with $N_A = 3$ in our example) buffers per incoming stream to ensure that

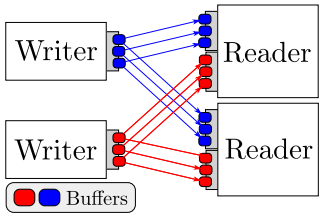


Figure 10.12: *Architecture of a VMPI_Stream.*

there is always a buffer available to receive any incoming data. This allows asynchronous reception of data blocks and without unexpected message as the MPI runtime is able to write directly in the reception buffer. On the opposite, on the writer side, N_A output buffers are shared between multiple endpoints, primarily to limit memory footprint (when using streams for instrumentation purpose, block size tends to be large $\approx$ 1 MB). VMPI_Streams can be created from a mapping in order to link two or more partitions, they can also be used between two arbitrary ranks. As a single stream can be connected to multiple endpoints, streams are initialised with a load-balancing policy which can be different at the two endpoints. Three basic policies are proposed : none, random, round-robin. Non blocking read is also supported to avoid deadlocks in multiple endpoints mode. When set, the call returns `EAGAIN` and tries the next endpoint according to the policy, avoiding circular waits.

VMPI_Stream Interface

Function	Description
VMPI_Stream_init	Initialises a VMPI_Stream with its block size and load-balancing policy.
VMPI_Stream_open	Open a stream to dest with a given mode (r,w,rw), should be called symmetrically.
VMPI_Stream_open_map	Open streams according to a mapping with a given mode.
VMPI_Stream_close	Close a stream and sends EOF to reading endpoints (if all streams closed).
VMPI_Stream_join	Merge two VMPI_Streams.
VMPI_Stream_read	Read from a stream a given number of blocks either in a blocking or in a non-blocking fashion.
VMPI_Stream_write	Write a given number of blocks to a VMPI_Stream.
VMPI_Stream_test_read	Returns true if at least one block can be read.
VMPI_Stream_test_write	Returns true if at least one block can be written.
VMPI_Stream_read_count	Returns the number of inbound streams.
VMPI_Stream_write_count	Returns the number of outbound streams.

Figure 10.13: *VMPI Stream interface.*

10.2.4 1 to N Coupling

The combination of virtualization, mappings and streams we briefly exposed, provides all the necessary components to perform runtime coupling. In order to illustrate our method, we perform the dynamic mapping shown in Figure 10.14. It consists in mapping N partitions to one — mapping strictly identical to the one we do when instrumenting multiple applications.

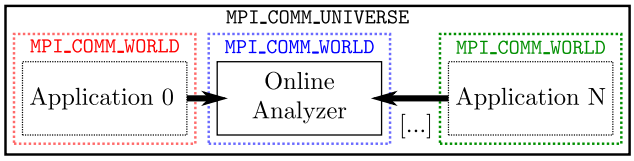


Figure 10.14: *Runtime coupling for multi-instrumentation purpose.*

The two source codes of Figure 10.15 and 10.16 are sufficient to build the runtime-coupling described in Figure 10.14. In this case, each application is transparently running in its sandboxed communicator thanks to virtualization. They are able to connect to each other using our mapping component and to setup inter-application communication channels which will adapt to available resources thanks to VMPI_Streams's load-balancing support. This example demonstrates that our approach efficiently handles communications and mapping from N to one partitions making multi-instrumentation trivial.

```
MPI_Init( &argc, &argv );
/* Fill in mapping data */
VMPI_Map map;
VMPI_Map_clear( &map );
/* Retrieve analyzer partition */
VMPI_Partition_desc *p_an =
                    VMPI_Get_desc_by_name("Analyzer");
/* Could not find analyzer */
if(!p_an ){
  printf("Could not locate analyzer partition\n");
  exit(1);
}
/* Map to analyzer */
VMPI_Map_partitions( p_an->id,
                    VMPI_MAP_ROUND_ROBIN, &map );
/* Setup Stream */
VMPI_Stream st;
/* Initialize stream */
VMPI_Stream_init( &st, 1024*1024,
                  VMPI_STREAM_BALANCE_ROUND_ROBIN);
/* Create streams according to mapping */
VMPI_Stream_open_map( &st, &map, "w" );
void *buff = malloc( 1024 * 1024 );
/* Send some data */
int i;
for( i = 0 ; i < 1024 ; i++ )
  VMPI_Stream_write( &st, buff, 1 );
/* Close Stream */
VMPI_Stream_close( &st );
free( buff );
MPI_Finalize();
```

Figure 10.15: *Code for a runtime-coupled instrumented program.*

10.2.5 Runtime-Coupling Performance

Figure 10.17 shows the throughput which can be achieved on Tera 100 with the coupling codes of Figure 10.15 and 10.16 for different $\frac{\text{writer}}{\text{reader}}$ ratios. The number of reader N_r for a given number of writer N_w is computed as follows : $N_r = \lfloor \frac{N_w}{\text{Ratio}} \rfloor$ if $1 < \lfloor \frac{N_w}{\text{Ratio}} \rfloor$, with a default value of $N_r = 1$ to make sure there is always one process reading. As it could be expected the best

```
/* Set partition name */
VMPI_Set_partition_name( "Analyzer" );
MPI_Init( &argc, &argv );
/* Fill in mapping data */
VMPI_Map map;
VMPI_Map_clear( &map );
int i, ret;
for( i = 0 ; i < VMPI_Get_partition_count(); i++){
  /* Map each partition except myself */
  if( i != VMPI_Get_partition_id() )
    VMPI_Map_partitions( i,
                         VMPI_MAP_ROUND_ROBIN,&map );
}
/* Setup Stream */
VMPI_Stream st;
/* Initialize stream */
VMPI_Stream_init( &st, 1024*1024,
                  VMPI_STREAM_BALANCE_ROUND_ROBIN);
/* Create streams according to mapping */
VMPI_Stream_open_map( &st, &map, "r" );
/* Allocate input buffer */
void *buff = malloc( 1024 * 1024 );
/* Start Read Loop */
do{
  /* Read one block of 1M from stream */
  ret = VMPI_Stream_read( &st, buff,
                          1, VMPI_STREAM_NONBLOCK);
  if( ret == VMPI_EAGAIN )
    continue;

  if( 0 < ret ){
    /* Process BUFFER */
  }
/* 0 means all remote streams are closed */
}while( ret != 0 );

VMPI_Stream_close( &st );
free( buff );
MPI_Finalize();
```

Figure 10.16: *Code for a runtime-coupled analyzer.*

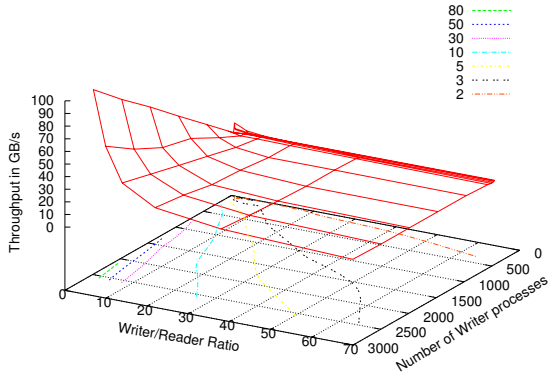


Figure 10.17: *Global throughput of VMPI_Streams when writing 1GB per process at various* $\frac{writer}{reader}$ *ratios with programs of figures 10.15 and 10.16.*

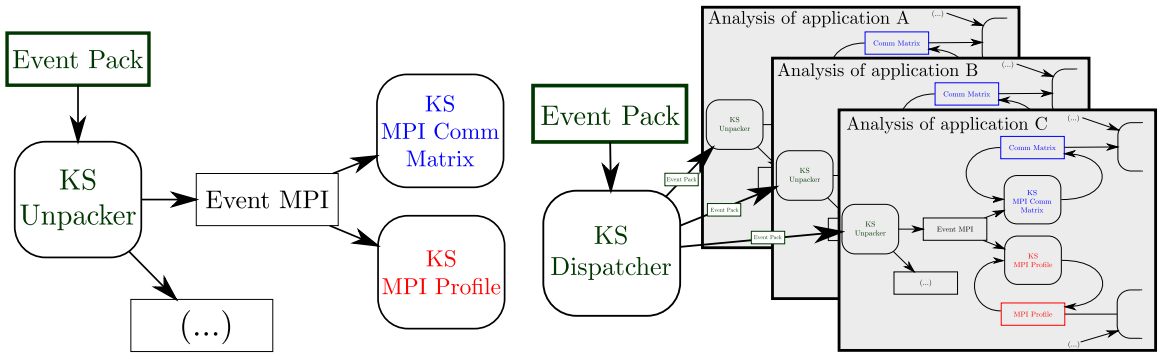
throughput is obtained when there are as many readers than writers, with for example at 2560 writers and readers, a cumulative throughput of 98.5 GB/sec between the two partitions. This value has to be compared with the maximum IO throughput of Tera 100 which is of 500 GB/s for the whole machine which, scaled back to 2560 cores and considering an even bandwidth balancing (expected because of the fat-tree topology), gives a theoretical throughput of 9.1 GB/s. Consequently, at this scale, VMPI_Streams are competitive with the file-system approach until a ratio of one reader for ≈ 25 writers. Practically, ratios between $\frac{1}{1}$ and $\frac{1}{32}$ provide enough bandwidth for profiling purpose, $\frac{1}{10}$ being a good bandwidth–resource trade-off.

10.2.6 Summary

This section introduced our coupling mechanism,we demonstrated how several applications can run concurrently (virtualization), connect to each other (mappings) and exchange data (streams). We also presented an example of N to one coupling, similar to what is done when instrumenting multiple applications. Eventually, we shown that our approach is competitive with file system in term of bandwidth. In the rest of this chapter, we will rely on this method to connect instrumentation and analysis completely bypassing file-system, and thus, both leveraging IO limitations and opening new parallelism opportunities. The rest of this chapter will introduce a parallel data-flow engine derived from Blackboard Systems which purpose is to perform an on-line performance data reduction.

10.3 Blackboard

Blackboard systems are good candidates to parallelise on-line reductions of large traces. They combine high performance with flexibility as several analysis can be chained opportunistically. This extends the approach previously described in section 9.5 by managing chained analysis while offering better parallelism. In our previous implementation, the maximum parallelism was the number of input files as reading threads were calling successive modules on each event. Whereas, thanks to the Blackboard model, parallelism can be extracted from events themselves. On-line coupling also enable new parallelism opportunities analysis can be pipe-lined — drastically reducing the time to result.



(a) Example of data-flow analysis.

(b) Concurrent application profiling with Multiple levels.

Figure 10.18: *Implementation of a data-flow analysis in a Blackboard System.*

Blackboard systems provide an anonymous storage structure. It allows cohabitation for chained analyses which are dedicated to a set of data types. Their processing are triggered each time suitable data are pushed on the blackboard. Our implementation relies on several worker threads, providing analysis with natural parallelism. Figure 10.18(a) presents a sample data-flow analysis implementation on a blackboard: event packs streamed from the instrumented application are 'pushed' on the blackboard, which triggers their unpacking by the 'KS Unpacker' which in turn pushes all the individual events. Then, MPI events are processed by both topological analysis and MPI profiler, in order to reduce events to their individual data-structure. This approach does not only simplify data-analysis expression, but also enables to analyse straightforward concurrent programs. At data-flow level, processing simply has to be replicated for each program, using multi-level blackboard, each level being dedicated to an application. As shown in Figure 10.18(b), a new KS in charge of dispatching each event pack to its associated blackboard level, provides a direct multi-instrumentation support. Knowledge sources can be developed in separated shared libraries which can be loaded dynamically, integrating new KSs on the blackboard. This simplifies the development of profiling modules as they can be developed in a very orthogonal manner. Moreover, thanks to the shared library mechanism, these KS can rely on third party dependencies such as image processing libraries and can perform analysis of arbitrary complexity with various levels of integration on the Blackboard. Modules can just refer to a single event for notification purpose or completely integrate themselves on the blackboard with multiple KS and data-types in order to benefit from data-flow parallelism.

10.3.1 Blackboard Implementation

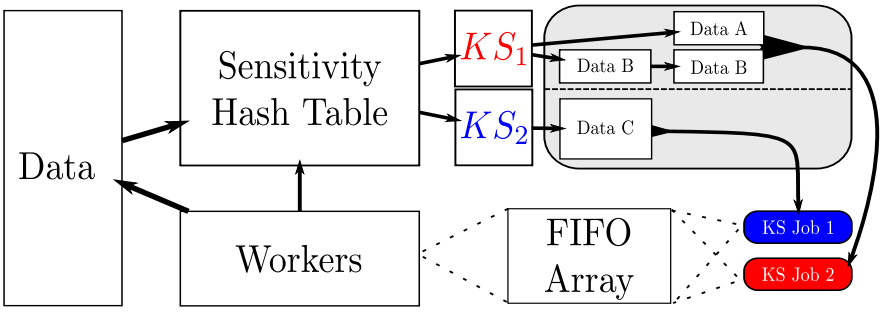


Figure 10.19: *Implementation of our BlackBoard architecture.*

Our blackboard implementation relies on two main components: data entries (DE) and Knowledge Sources (KS). A Data Entry can be defined as a tuple: $\{\mathtt{Type}, \mathtt{Size}, \mathtt{Payload}\}$ with *Type* being an integer identifier, *Payload* an arbitrary blob of data the size of which is given by *Size*. A knowledge source is a couple: $\{\{\mathtt{Sensivities}\}, \mathtt{Operation}\}$ with *Sensitivities* being a set of $\{\mathtt{Types}\}$ triggering an *Operation* defined as a function called over the input data. In order to keep the control system as simple as possible, operations are first described as a static data-flow. A simplified form of opportunistic reasoning is provided by the ability of any KS to register or remove any KS including itself. The Control System (see Figure 6.4) is only in charge of triggering KS with satisfied sensitivities. Figure 10.19 presents the architecture of our parallel blackboard: when a data entry is submitted, matching sensitivities are looked up in the sensitivities hash table; if a matching KS is found, a reference to the data entry is

pushed in a FIFO. In the case it is last unsatisfied sensitivity, a new job is created as a couple $\{\{Data\ entries\}, Operation\}$. In order to reduce contention, jobs are randomly pushed in an array of FIFOs, individually protected by a lock. A pool of workers is constantly looking for jobs by sweeping FIFOs from a random starting point while a back-off system prevents threads from spinning over the locks in the absence of jobs. In order to make parallelism manageable, data entries are generally read only and are managed using a ref-counting mechanism, a data being writable only if its ref-counter is equal to one. Besides, when data are pushed on the blackboard, they are stored in a buffer which is automatically freed after all the processings linked to this data are done. This allows the use of the blackboard as a temporary storage medium, freeing MPI_Streams's communication buffers and avoiding to block instrumented processes. The multi-level blackboard is implemented using data-entries identifiers which are computed as a hash of both level and data-type names. By doing so, identical KSs and data-types can be present in multiple blackboard levels, providing multi-analysis as depicted in Figure 10.18(b).

Blackboard Interface

Function	Description
MALP_blackboard_init	Initialise a Blackboard with a given number of workers.
MALP_blackboard_release	Release a blackboard.
MALP_blackboard_set_default_KS_handler	Set a default handler for data blocks with no associated KS.
MALP_blackboard_wait	Wait for all data entries to be processed.
MALP_blackboard_wait	Wait for all data entries to be processed.
MALP_Blackboard_ks_present	Returns true if KS of a givent type is present.
MALP_Blackboard_new_ks	Register a new Knowledge Source.
MALP_Blackboard_delete_ks	Delete one KS of a given type.
MALP_Blackboard_submit_data	Submit a data entry of a given type (from a memory segment).
MALP_Blackboard_submit_data_entry	Submit an existing data entry.
MALP_Blackboard_writable_data	Returs true if this entry is a data entry is writable.
MALP_Blackboard_pending	Returns the number of data entries pending in the BB.
MALP_Blackboard_wait_pending	Wait until N data entries are pending.

Figure 10.20: *Parallel Blackboard interface.*

10.3.2 Limitations

As discussed in previous sections, our coupling framework is built over MPI in MPMD mode, despite allowing direct integration in existing supercomputers' batch managers and software stack, this design choice has drawbacks. Currently, a user who wants to instrument a set of programs using our tool has to launch a job consisting in his own programs plus the distributed analysis engine. Resources being statically assigned, if programs have very different wall-times, analysis resources will be over-sized for a part of the execution. A solution to overcome this problem would be MPI dynamic processes, but they are not particularly easy when spawning multiple communicating processes. Even if we deal with the complexity of MPI dynamic processes, we will not be able to provide an inter-node instrumentation service, because ideally, it should not be included in the batch manager but instead in an identified set of nodes providing a service. An implementation at network layer level would provide more flexibility with dynamic program registration, persistence of instrumentation servers, and more opportunities to manage authentication and centralisation of profiling metrics.

Dealing with analysis, our current blackboard implementation only provides analysis within nodes boundaries, preventing analysis dependent from a global state. This limitation is comes from the fact that our blackboard is not distributed across nodes, preventing remote queries. As presented in next chapter we plan to address this limitation by extending out blackboard with a communication engine. Allowing analysis to be distributed, and thus opening possibilities for more interesting analysis.

10.3.3 Summary

This chapter presented MALP which is the successor of the MPC Trace library. It has been completely rewritten to leverage IO limitations encountered by our first implementation which relied on an unacceptable number of files. We first introduced our coupling method which separates (virtualization), associates (mappings) and connect (streams) groups of processes. Then, by coupling this communication primitive with a parallel data-flow engine inspired from Blackboard systems, we demonstrate how concurrent analysis of MPI programs can be achieved. Our pipe-lined on-line analysis approach is therefore a method which both leverages IO limitations and opens new parallelism opportunities. Approach providing an improved scalability when compared to our first implementation while reducing the time to result thanks to its on-line nature.

CHAPTER 11

Distributed Analysis and Reduction Tree (DART)

This chapter introduces the Distributed Analysis and Reduction Tree (DART) which is a trial to extend the parallel blackboard previously developed for our on-line analysis (in MALP) in purpose of building a distributed one. We first detail the reasons which motivated this development before introducing the architecture we adopted. Then, we detail our current interface and explain how analysis could benefit from such infrastructure. Eventually, we detail limitations which prevented us to stabilise its implementation before the redaction of this document while pointing out aspects which still have to be enhanced to reach our goal.

11.1 Motivations

The parallel blackboard architecture we introduced in Section 10.3 provides efficient data-flow analysis at node level thanks to shared memory parallelism. Despite this approach offers enough semantic to reduce profiling data in place, generating various profiles (as presented in section 12.4), local reduction alone cannot be satisfactory. Indeed, several analysis have to cross analysis node boundaries, for example, to collate distributed events (point to points, one sided, collective) to derive performance metrics similar to those generated by Scalasca about wait-states [BGWA10]. In addition, from a data-flow aspect, it could be useful to be able load balance or sort the profiled data, structuring analysis in sub-parts or specialised units, for example for distributed phase analysis. Moreover, when running on a massively parallel machines, our current blackboard implementation is very frustrating as it is not able to cross nodes boundaries, and therefore has to rely on MPI to perform the final reduction, making it is some aspects very close to a basic MapReduce approach as performance data are sent to buckets (Map, via VMPI_Maps) in order to be processed locally (temporal reduction over events) before being reduced (MPI Reduce) spatially, producing the final result which is converted in a latex report. Consequently, our approach needed further development to allows non-structured analysis which preserve the opportunistic reasoning aspect of Blackboard systems. This led us to the development of DART during the last month of this thesis in order to overcome these limitations by building a fully distributed blackboard with support for hybrid parallelism, allowing more ambitious analysis.

11.2 Architecture

DART is build over both our coupling mechanism (see Section 10.2) and a parallel Blackboard (see Section 10.3). It augments the blackboard with both a topology and communication primitives in the purpose of allowing distributed analysis. This section is mainly focused on DART communication engine which is at the core of DART's features.

11.2.1 Fixed Topology

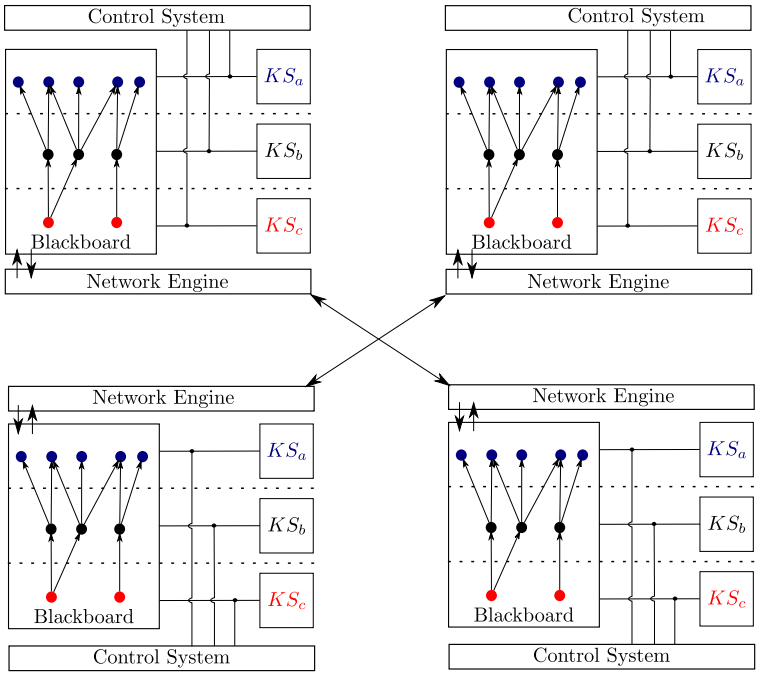


Figure 11.1: *Overview of the distributed Blackboard architecture.*

As presented in figure 11.1, a distributed blackboard is built from several blackboard interconnected thanks to a network engine. In DART, the network has a known topology in order to define a clear neighbouring of processing units and therefore, spatially locating the computation. As we explained when we introduced the MapReduce principle (see Section 6.4.3), one challenge when dealing with distributed computation is to manage data scattering and design scalable spatial interactions. Process which can become challenging in MPI where point to point communication have to be defined from both endpoints (pair of MPI_Recv and MPI_Send), possibly leading to a parallelism loss if computation is imbalanced. Therefore, in our distributed blackboard approach we rely on a one-sided approach, avoiding task coupling through communications. This approach is not new as it is at the basis of Partitioned Global Address Space (PGAS) or Distributed Shared Memories (DSM) which allows any process/node to access sections of a global memory which is partitioned over several nodes — completely removing the need for explicit communications. However, implementing a PGAS is challenging task as it requires complex coherency [LH89, NL91] algorithms and have to cope with some limitations of the underlying MPI run-time, particularly with Remote Memory Accesses (RMA) [BD04].

In DART, transfers are explicit but only defined from the sender side, this poses the problem of memory exhaustion on the receiver side which can possibly be flooded during N to one patterns. To face this problem, we counted on the limited asynchronous window of VMPI_streams which behave like UNIX pipes, blocking if data are not consumed while preserving data ordering. Then, instead of directly sending data to the target Blackboard, we rely on a routing network to reach neighbouring Blackboards which are implicitly aware of remote blackboard availability, simply by checking if their stream is ready for writing — this process can be described as spatial spilling. Consequently, this N to one pattern becomes a succession of pending data-blocks in remote processes memory, all directed to the target process. Naturally, this process is clearly more costly than the direct network interfacing but is a simple way to prevent both resource exhaustion and ordering problems. Solution which is quite simple, particularly when compared to solutions relying on one sided communications, approach which generally requires handshakes [HWM02, WR07].

Topology Calculation

The first requirement to build our routing network is the capacity of efficiently computing the topology. To simplify the routing, we relied on d-meshes from dimension 1 to 3. In this purpose, we rely on a splitting of the binary representation over each dimension in order to map a linear identifier into a topological space.

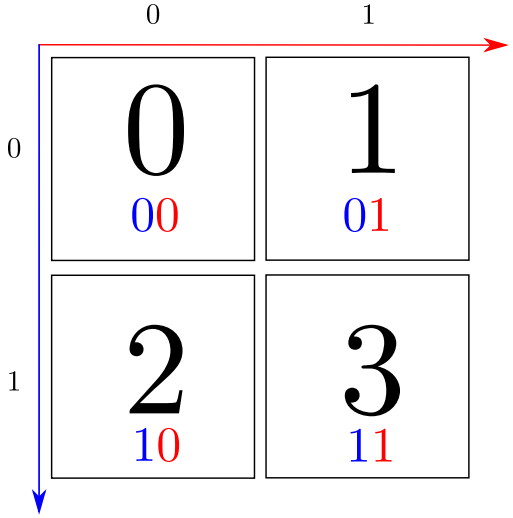


Figure 11.2: *Simple example of 2D topology computed from the binary representation.*

Consequently, the mapping process presented in figure 11.2 can be generated as follows:

- Compute the number of bits needed to store N values (in our case $N = 3$) by incrementing p in $2^p - 1$ until this values is equal or higher than N. This immediately yields $p = 2$.
- In a second time, we compute b_{dim} the number of bits per dimension such as $b_{dim} = \frac{p}{d}$ with d the dimension of the mesh (here $d = 2$). We store this value in a tuple of dimension d which we call *width*, while taking care of storing the remainder in the last dimension (p is possibly not divisible by d).

- Then we repeat the same process in terms of number of nodes in order to determine the remaining processes in the last dimension, those values are stored in a tuple called *ceil*.
- Eventually we compute the *masks* and *shifts* associated with each dimension. The mask is simply $width$ bit up and the shift is the sum of the previous dimension width.

Therefore, if we consider the mapping of 256 Blackboards in a 3D-mesh we get:

$$
\begin{aligned}
&(1111\ 1111)_b = (255)_d \text{ (p=8 bits)}\\
&d = 3\\
&b_{dim}(1) = 3,\quad b_{dim}(2) = 3,\quad b_{dim}(3) = 2\\
&Ceil_{dim}(1) = 8,\quad Ceil_{dim}(2) = 8,\quad Ceil_{dim}(3) = 4\ (8 \times 8 \times 4 = 256)\\
&(11\ 111\ 111)_b\\
&Mask(1) = (00\ 000\ 111)_b,\ Shift(1) = 0 \qquad (11.1)\\
&Mask(2) = (00\ 111\ 000)_b,\ Shift(2) = 3 \qquad (11.2)\\
&Mask(3) = (11\ 000\ 000)_b,\ Shift(1) = 6 \qquad (11.3)
\end{aligned}
$$

Yielding the following topology (output of our topology module) when connected in 8-neighbour (including diagonals):

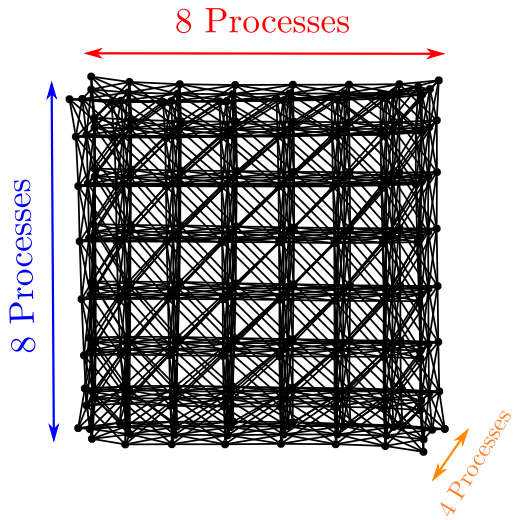


Figure 11.3: *3D-mesh topology for 256 processes.*

Dealing with identifier to coordinate conversions, we rely on mask and shift tuples which provide straightforward conversions. For example if we are looking for the coordinate (x, y, z) of the Blackboard with the identifier 123 it the 3D-mesh we have just computed:

$$
\begin{aligned}
&(0111\ 1011)_b = (123)_d\\
&(01\ 111\ 011)_b\\
&x = ((123)_d\ \&\ Mask(1)) >> Shift(1) = ((123)_d\ \&\ (00\ 000\ 111)_b) >> 0 = 3\\
&y = ((123)_d\ \&\ Mask(2)) >> Shift(2) = ((123)_d\ \&\ (00\ 111\ 000)_b) >> 3 = 3\\
&z = ((123)_d\ \&\ Mask(3)) >> Shift(3) = ((123)_d\ \&\ (11\ 000\ 000)_b) >> 6 = 1\\
&(x, y, z) = (3, 3, 1)
\end{aligned}
$$

A similar process can be derived in order to compute the identifier of any coordinate. For example if we now look for the Blackboard with coordinate $(x = 1, y = 2, z = 3)$:

$$
\begin{aligned}
x &= (1)_d = (00\ 000\ 001)_b \\
y &= (2)_d = (00\ 000\ 010)_b \\
z &= (3)_d = (00\ 000\ 011)_b \\
id = \ & (x << Shift(1)) \ \& \ Mask(1) \\
| \ & (y << Shift(2)) \ \& \ Mask(2) \\
| \ & (z << Shift(3)) \ \& \ Mask(3) \\
id = \ & ((00\ 000\ 001)_b << 0) \ \& \ (00\ 000\ 111)_b \\
| \ & ((00\ 000\ 010)_b << 3) \ \& \ (00\ 111\ 000)_b \\
| \ & ((00\ 000\ 011)_b << 6) \ \& \ (11\ 000\ 000)_b \\
id = \ & (11\ 010\ 001)_b = (209)_d
\end{aligned}
$$

Neighbouring Calculation

During an actual execution, the whole graph such as in Figure 11.3 is never actually stored as it would be too expensive for larger runs where the adjacency matrix would quickly grow ($\Theta(N^2)$ with N the number of Blackboards). Instead, each separated processing unit has to derive its neighbouring, allowing a constant sized storage. This section describes the process which allows the computation of the neighbours for d-meshes or tori with a lightweight data structure. In the rest of this section we will exemplify this computation on the 2D-mesh of figure 11.4 which has been generated in a 4-neighbour configuration (no diagonals).

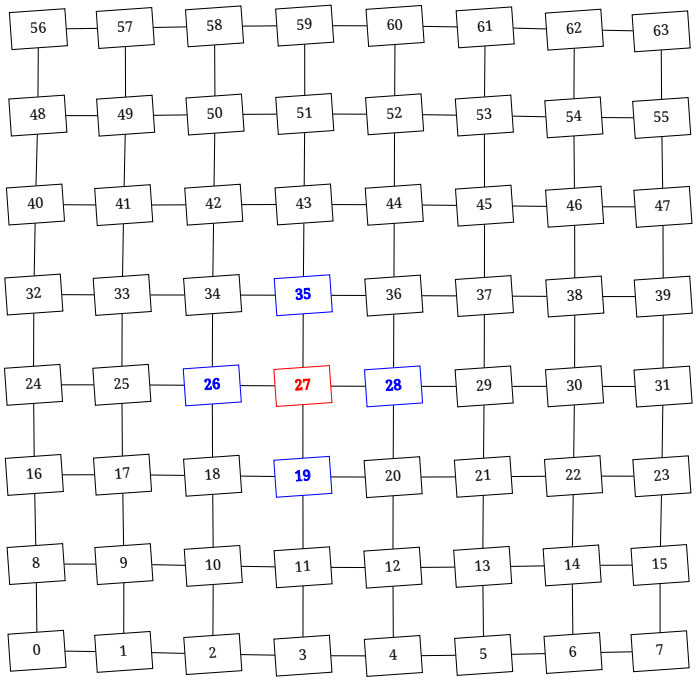


Figure 11.4: *2D-mesh connected with 4-neighbouring of 64 nodes with node 27 in red and its neighbouring (19,26,28,35) in blue.*

This 2D-mesh of 64 nodes can be described with two dimensions of three bytes $(111\ 111)_b$, yielding directly the following shifts and widths:

$$\begin{aligned} Mask(1) &= (000\ 111)_b, \ Shift(1) = 0 \\ Mask(2) &= (111\ 000)_b, \ Shift(2) = 3 \end{aligned}$$

Allowing us to compute the coordinates (x, y) of the node 27:

$$\begin{aligned} (27)_d &= (011\ 011)_b \\ x &= (011)_b = (3)_d \\ y &= (011)_b = (3)_d \\ (x, y) &= (3, 3) \end{aligned}$$

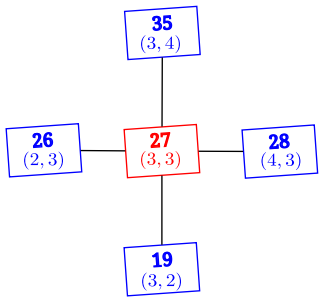


Figure 11.5: *Detail of node 27's neighbouring with associated coordinates.*

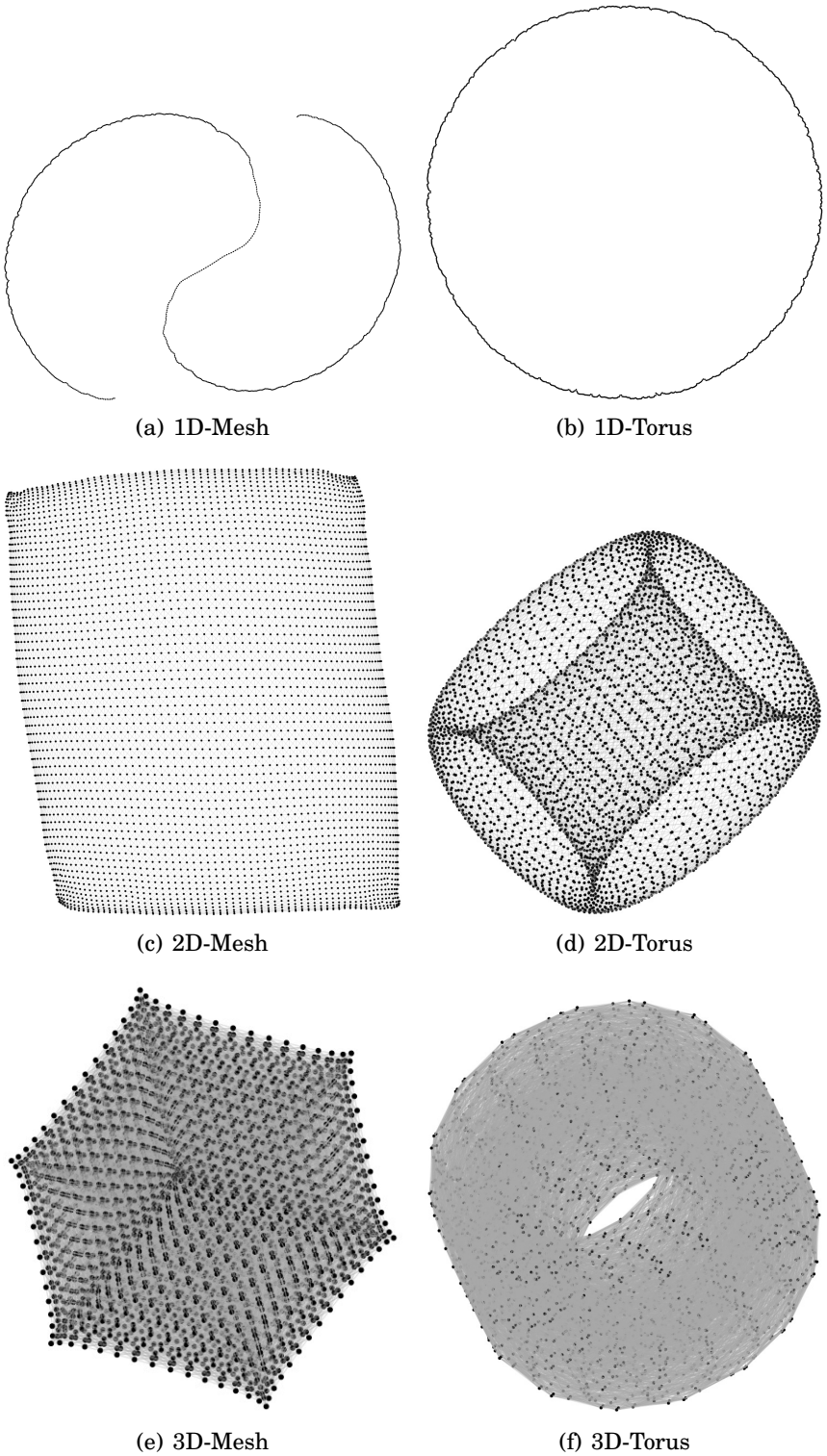
By looking at figure 11.5, it can be seen that neighbours are simply derived by incrementing and decrementing each coordinate while taking care of remaining in the limits of the mesh:

$$\begin{aligned} (3,4) &= ((011)_b, (100)_b) \rightarrow (100\ 011)_b = (35)_d \\ (4,3) &= ((100)_b, (011)_b) \rightarrow (011\ 100)_b = (28)_d \\ (3,2) &= ((011)_b, (010)_b) \rightarrow (010\ 011)_b = (19)_d \\ (2,3) &= ((010)_b, (011)_b) \rightarrow (011\ 010)_b = (26)_d \end{aligned}$$

Moreover, in order to generate a torus, if the neighbour is over the ceiling or lower than 0, the dimension is 'wrapped around' in order to produce a torus (wrapped around mesh). This process can be repeated to generate various configurations of meshes as presented in figures of Figure 11.6 which were generated using the Graphviz [GN00] graph visualisation tool, illustrating the variety of topologies which can be generated using this simple approach.

Routing

As far as the routing is concerned, we also use a simple method which relies on the quadratic distance: for each neighbour, we compute its quadratic distance with the destination and choose the nearest. We rely on the quadratic distance as computing the effective distance



(a) 1D-Mesh (b) 1D-Torus

(c) 2D-Mesh (d) 2D-Torus

(e) 3D-Mesh (f) 3D-Torus

Figure 11.6: *Example of meshes (wrapped around or not) with 4096 nodes as generated using the method we introduced in this section.*

would require a square-rooting which is quite expensive and useless as the square-root function is monotonous and what we are interested in here is to find the nearest node, not a distance. We therefore rely on formula 11.4 which computes the distance between two nodes X and Y in a d mesh. Note that we limited our routing algorithm to unwrapped meshes as we faced routing problems using the torus distance in presence of multiple paths which led to the formation of loops when using this (too?) simple approach.

$$D(X,Y)^2 = \sum_{j=1}^{d} (X_j - Y_j)^2 \tag{11.4}$$

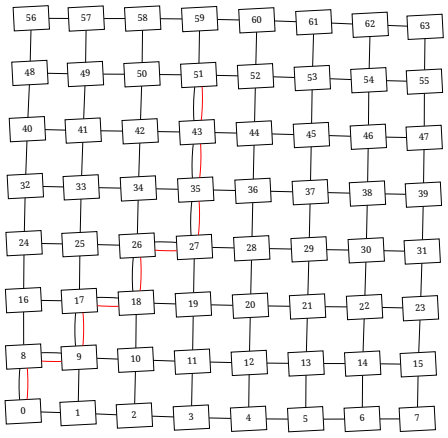


Figure 11.7: *Sample route (in red) from node 51 to 0 as computed using this distance.*

11.2.2 Network Engine

Now that we introduced our topology management method and our simple routing algorithm, this section introduces the *Network Engine* we added to our Blackboard system in order to build a distributed Blackboard.

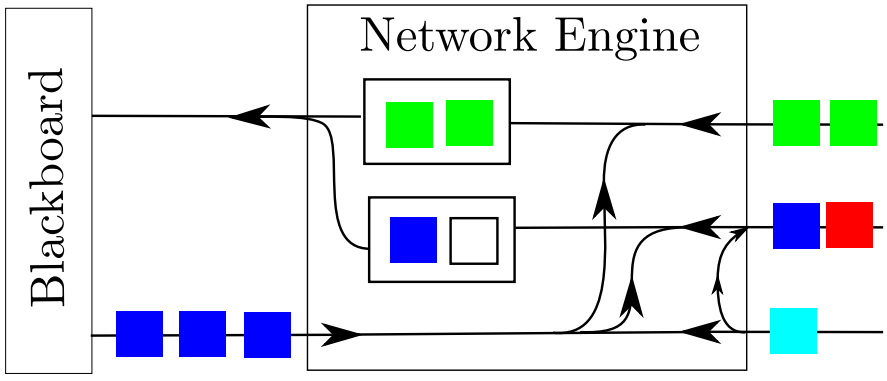


Figure 11.8: *Overview of the Network Engine.*

As presented in figure 11.8, our network engine is relies on a single threaded progress loop in order to remain compatible with MPI runtimes which do not support the THREAD_MULTIPLE level of parallelism. Therefore, messages are sent from the Blackboard through a FIFO which is regularly scrutinised in search of message to send. Dealing with incoming streams the same process takes place in round robin between streams. Sending an receiving phases are allocated equivalent quantum of time in terms of a maximum number of operations and are executed alternatively. Dealing with the data layout itself, we rely on small data-blocks (1KB) which were inspired from flow control digits (flits) in high speed networks. Indeed, it is important to split messages in order to balance the routing effort on several streams instead of monopolising the router for a large message while blocking other streams. Store-and-forward which consists in receiving the whole message before sending it is a very inefficient approach which monopolises memory on the nodes, and augments latency. Therefore, splitting the message in packets in crucial to gain in efficiency (see [Gra03] Section 2.5).

Data blocks are gathered on a per source basis as we assume that a given node will send messages in a partial order relatively to a single destination, guaranteeing that another message wont begin before the previous has been fully assembled. Once assembled the data block is directly pushed on the blackboard with a type which has been encapsulated in the message by the sender. Consequently, thanks to this mechanism, Blackboards are able to push data remotely, triggering the associated processing with no need for explicit synchronisation (such as a paired receive or connection). Eventually, it shall be noted that each Blackboard has a parent, organising nodes in a binary tree shape which can be useful for reduction patterns (see next section).

11.3 Interface and Programming Principle

Function	Description
Dart_Engine_init	Initialise the DART engine with a given topological policy
Dart_Engine_release	Release the DART engine after waiting for job termination
Dart_Engine_push	Push a new data entry in the Blackboard
Dart_Engine_send	Send a data entry to a remote Blackboard according to its identifier
Dart_Engine_send_to_parent	Send a data entry to parent blackboard (according to a binary tree topology)
Dart_Engine_register_KS	Register a new Knowledge Source to process a given set of datatypes
Dart_Engine_register_reduction	Register a reduction for a given datatype.

Figure 11.9: *Distributed Blackboard interface.*

Knowledge System Registration

As presented in figure 11.9, DART's interface is relatively compact and consists in five calls, providing all the data managements primitive to implement a distributed Blackboard-inspired data-flow analysis. As previously with the parallel Blackboard, Knowledge Source can be registered at any moment on a set of data-types with a call to `register_KS` which has the following footprint:

```
void Dart_Engine_register_KS( struct Dart_Engine *dart_engine, void (*handler)(), char *names, ...)
```

This function call takes the Dart_Engine in parameter a pointer to function and a list of data-types defined as strings. A sample registration of a KS over two data-types A and B can be defined as follows (NULL is used to delimit the variadic argument list):

```
void KS_A_B( struct Dart_Engine *de,
             struct MALP_blackboard *bb,
             struct Data_entry *De_A, struct Data_entry *De_B) {
  /* Process Data Entries
     Push new data entries locally or remotely
     Or register new KSs */
}
[...]
Dart_Engine_register_KS( &dart_engine, KS_A_B, "A", "B", NULL );
```

Data Entry Submission

Then, each time two data entries A and B are pushed on the Blackboard the `KS_A_B` function is called with those to entries in parameter. Named data entries can be pushed as follows:

```
struct A_entry A;
struct B_entry B;
Dart_Engine_push( &dart_engine, "A", &A, sizeof(struct A_entry) );
Dart_Engine_push( &dart_engine, "B", &B, sizeof(struct B_entry) );
```

Similarly, data-entries can be created locally without being directly pushed on the Blackboard, this behaviour is useful for example when pushing a data on a remote Blackboard. It uses a syntax very similar to `Dart_Engine_push`:

```
struct Data_entry * Dart_Engine_new_data( struct Dart_Engine *dart_engine,
                                          char *name, void *payload, size_t size )
```

Once stored as a `Data_entry` it is possible to push it on a remote Blackboard allowing the creation of distributed data-flow using the following call:

```
void Dart_Engine_send( struct Dart_Engine *dart_engine,  struct Data_entry * entry, int dest );
```

Which can be used as follows:

```
struct A_entry A;
struct Data_entry * De_A = Dart_Engine_new_data( &dart_engine, "A", &A, sizeof(struct A_entry) );
void Dart_Engine_send( &dart_engine, De_A, 1 /* DEST  (MPI Rank) */ );
```

Another primitive similar to `Dart_Engine_send` can be used to send a `Data_entry` to parent process according to a binary tree topology, if called on the root, this call pushes the data on the local Blackboard:

```
void Dart_Engine_send_to_parent( struct Dart_Engine *de,  struct Data_entry * entry )
```

Reduction Implementation

As depicted in figure 11.10, our reduction relies on a binary tree which is defined through parent, children relationships. An user can register a reduction for a given data-type with a single call to:

```
void Dart_Engine_register_reduction( struct Dart_Engine *de,
                                     char *input_name,
                                     void (*red_func)(struct Data_entry *new_de,
                                                      struct Data_entry **entries, uint32_t count )
                                   )
```

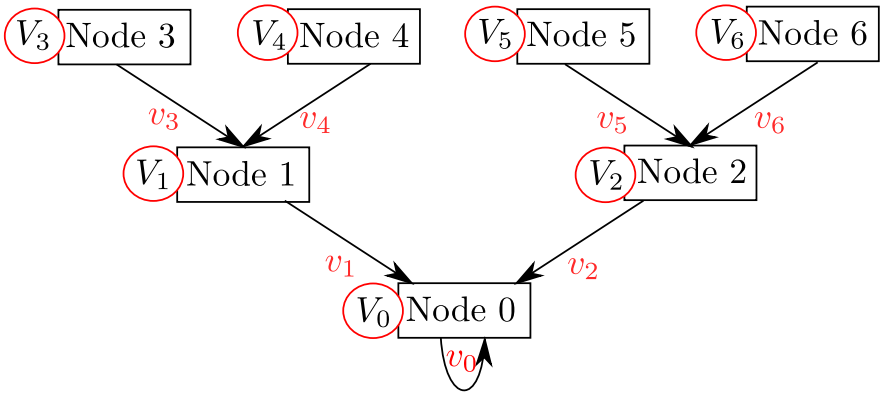


Figure 11.10: *Overview of the reduction process.*

This call will register a reduction for a type `input_name` which will be reduced by a function `red_func` before being sent to the parent (or local Blackboard if root). If we look with Figure 11.10 at the reduction process itself, we can derive partial equations which eventually can be used transitively in order to build the whole sum in a very straightforward way:

$$
\begin{aligned}
v_3 &= V_3, \quad v_4 = V_4 \\
v_5 &= V_5, \quad v_6 = V_6 \\
v_1 &= V_1 + v_3 + v_4 = V_1 + V_3 + V_4 \\
v_2 &= V_2 + v_5 + v_6 = V_2 + V_5 + V_6 \\
v_0 &= V_0 + v_1 + v_2 \\
v_0 &= V_0 + V_1 + V_3 + V_4 + V_2 + V_5 + V_6
\end{aligned}
$$

Consequently, when implementing a reduction KS, we have to setup sensitivities to the data-type which comes from each child and a local data-entry. This is done using the command `DART_KEY_FROM( a , src )` which relies on a sub-part of the data-type identifier (hash of name) which can be used to specify up to ($0xFFFF = 65535$)[1] source, allowing the query data not only from their type but also from their source. Aspect which is important when processing reduced data types which are dependent from where data are coming from. Once the result reaches the root node it is pushed locally with the name `Red(T)` with `T` the type of the reduced data-entry. A sample reduction using this mechanism is presented in figure 11.11.

[1] This limitation shall be removed in later implementations by using larger identifiers to describe data-types.

```c
#include <DART_Engine.h>

/*#### Print the value on 0 once reduced  ####*/
void print_red_double_handler( struct Dart_Engine *de,
                               struct MALP_blackboard *bb,
                               struct Data_entry *a_double) {
  double *in = (double *)a_double->payload;
  printf("Reduced %g\n", *in );
}

/*#### Perform the reduction ####*/
void reduce_double(struct Data_entry *new_de, struct Data_entry **entries, uint32_t count) {
  /* Initialise the output buffer */
  double *out = (double *)new_de->payload;
  *out=0;

  /* Perform the reduction over count values (including local one) */
  int i;
  for( i = 0 ; i < count ; i++ ) {
    double *in = (double *)entries[i]->payload;
    *out += *in;
  }
  /* Out is automatically sent to parents by callee */
}

/*#### Main Function ####*/
int main( int argc, char **argv ) {
  /* Initialise MPI context */
  int d;
  MPI_Init_thread( &argc, &argv, MPI_THREAD_MULTIPLE, &d);
  int rank, size;
  MPI_Comm_rank( MPI_COMM_WORLD, &rank);
  MPI_Comm_size( MPI_COMM_WORLD, &size);

  struct Dart_Engine de;
  /* Setup the topology as 2-tree (could have been a mesh)
   * However any topology at least embeds a 2-tree
   * for reduction purposes */
  struct Policy tree;
  Policy_init_tree(&tree, rank, size);

  /* Initialise the DART engine with the given topology */
  Dart_Engine_init( &de, &tree );
  /* Register the reduction handler */
  Dart_Engine_register_reduction( &de, "double",  reduce_double);
  /* Register the handler which will process the result 'Red(double)'*/
  Dart_Engine_register_KS( &de, print_red_double_handler, "Red(double)", NULL );

  double val = 1;
  int i = 0;
  /* Push values on local Blackboards for streamed  reduction */
  for( i = 0 ; i < 123456 ; i++ )
    Dart_Engine_push(&de, "double", (void *)&val, sizeof(double) );

  /* Wait for the result to be displayed and release */
  Dart_Engine_release( &de );

  /* Free MPI environment */
  MPI_Finalize();
  return 0;
}
```

Figure 11.11: *Sample streamed-reduction of several* `double` *implemented in DART.*

11.4 Analysis Projects

Now that we have introduced the global principle of the distributed Blackboard, we introduce two types of analysis which we are studying at this state of the thesis. We start by a continuous sampling engine before before presenting phase based sorting filter.

11.4.1 Continuous Sampling Engine

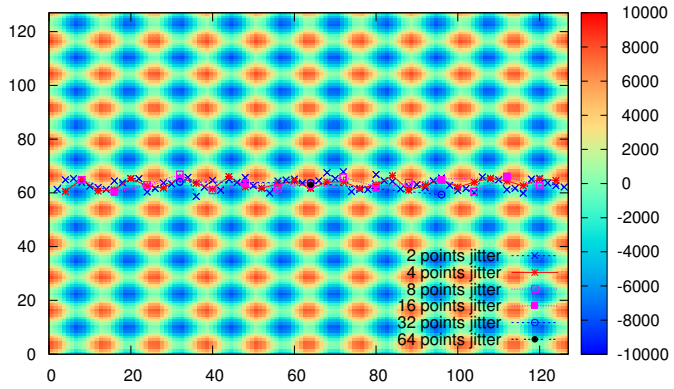
If we consider the parallel computation as a whole, it could be interesting to derive real time values such as memory communication, IO bandwidth or floating point operations per second (flops). To do so, we need to design a mean of continuously reducing sampled values in space over a time axis. We can model this process as the projection of N two dimensional values over a single two dimensional value. To do so we simply propose to use the barycentre definition where the barycentre $B(V(x,y))$ of a discrete set of N points P with coordinates (x_i, y_i) and weighted with $V(p_i)$ is such as:

$$B(P) = (x_b, y_b) = \frac{1}{\sum V(P_i)}(\sum V(P_i) \times x_i, \sum V(P_i) \times y_i) \tag{11.5}$$

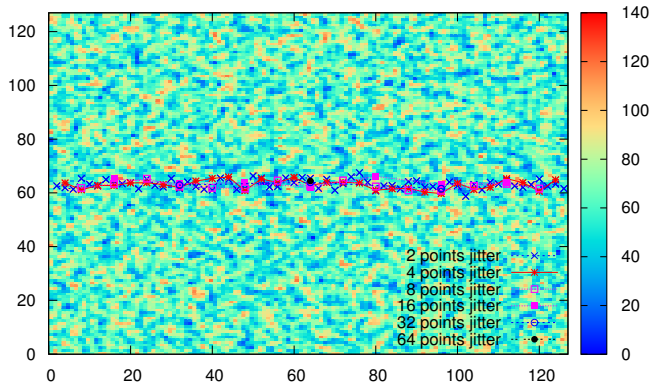
$$V(B(P)) = \overline{V(P_i)} = \frac{\sum V(P_i)}{N} \tag{11.6}$$

Formula which allows us to project a set of points to its barycentre (which coordinates are given by Equation 11.5) , assuming it has the average value of the set (Equation 11.6). We now propose to use this approach in the context of a reduction which samples several values in a given time-frame over several processes. As there can be a temporal jitter, in reduction, it is interesting to compensate it while privileging larger values in order to find an approximated position in both space and time for this set of samples. Moreover, thanks to barycentre associativity (the barycentre of two sets of two points is the barycentre of their barycentre, and so on ...) we can compute this value using a steamed-reduction similar to the one described in Figure 11.11, greatly simplifying the implementation of this analysis.

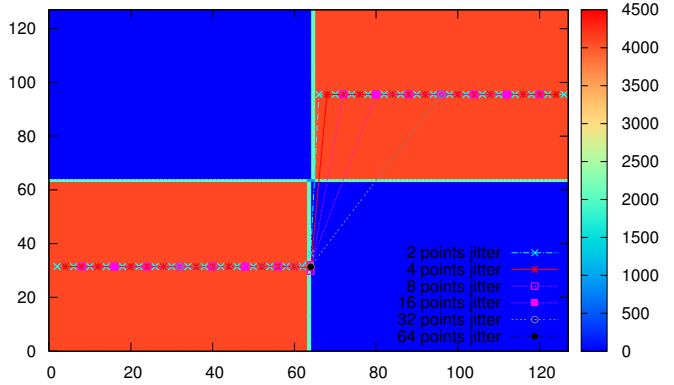
As presented in Figure 11.12, this method can be used to spatially describe a measure over a set of processes. Using this method the average value is mapped at a position which matches the barycentre of samples. As depicted in Figure 11.12, which illustrates this method on sample data-sets with different random jitters (on the x axis), if value patterns are either regular (Figure 11.12(a)) or random (Figure 11.12(b)), processes regularly distributed among values, leading to a relatively stable barycentre. However, if the behaviour is structured, with either square signals (Figure 11.12(c)) of sinusoidal waves (Figure 11.12(d)), we can observe that we capture the average behaviour. Moreover, if we look a the different jitter rates of Figure 11.12, we can see that the overall behaviour is still captured despite a temporal uncertainty. Consequently, we have seen that this relatively simple method can be applied to our streamed-reduction approach in order to generate a continuous flow of functioning points. Points which can be used to describe in some extent the average process behaviour not only on parameter side but also spatially. We plan to use this approach in higher dimension by faking the topology of processes in order to get a more discriminating barycentre, value which could be represented using series of radar charts.



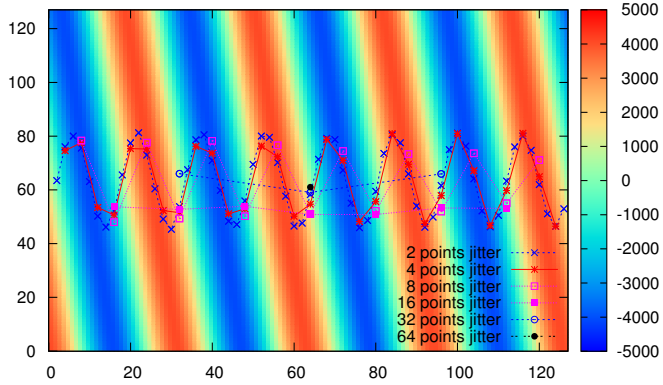
(a) Regular pattern.



(b) Random pattern.



(c) Symmetrical square signal.



(d) Diagonal waves.

Figure 11.12: *Example baricentre signal generated using Equations 11.5 on synthetic performance event signals.*

11.4.2 Phase Based Sorting Filter

If we consider the source code of a program it could be interesting to let the programmer describe hierarchical phases which could describe various computations steps. For example, for a simulation time-step: CFL computation, ghost-cell handling, communications, fluxes computation and time-stepping. One problem when managing a large number of phases is that we have to handle them in each analysis process, possibly leading to a redundancy in accumulation tables and eventually saturating memory. Consequently, it would be interesting to perform a spatial splitting at phase level in the analysis nodes. This means that we could preform the global analysis in every analysis nodes and then perform phase based analysis in a subset of nodes to which events would be rerouted.

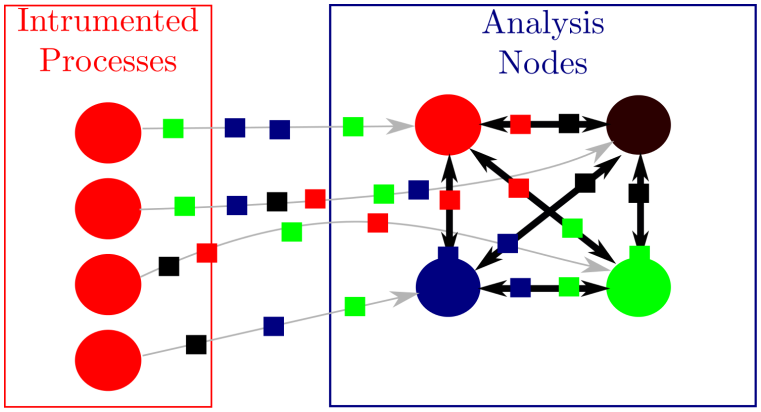


Figure 11.13: *Example of phase based spatial splitting of analysis.*

Consequently, events could be sent from the instrumented application to the analyser using the whole bisection bandwidth, then internally to the analyser, groups of processes, could exchange events in order to route phases related events to the correct processes, using a sub-part of the overall internal bandwidth budget. Approach which would allow a larger number of phases, as the would be distributed on sub-sets of nodes instead of impacting indifferently all processes. Indeed, for example, if we consider that a phase analysis (all the accumulation tables, hash tables which reduce events from a given set of processes, see coupling topology Section 10.2.2) takes a volume of memory V, in a node which has M memory, when can have n analysis with $n = \frac{M}{V}$. But now if we split this phase space over p processes, we get p memory spaces of size M each able to store $n = \frac{M}{V}$ phase analysis, leading to a gain of p in the number of phases which can be processes, thanks to this spatial reduction.

11.5 Limitations

We have seen that DART relies on a single thread to perform communications as it has been implemented over Bull MPI which is derived from OpenMPI and has a limited support for `THREAD_MULTIPLE`, leading to random crashes (when running with several threads). Therefore, messages are sent by a single thread which receives messages from the Blackboard via a FIFO, ensuring a certain form of fairness and limited asynchronism relatively to the communication engine. However, this approach, made compulsory by the underling MPI implementation is clearly impacting our overall performance, serialising communication operations in

both directions. A solution to this limitation would be to rely on the MPC framework which has been built to support programming model mixing and has a full `THREAD_MULTIPLE` support which would allow Knowledge Source to directly send data while preserving parallelism. Morevoer, MPC has been designed to run with a very large number of threads which could be used by the parallel blackboard.

Another aspect which was problematic was the routing in torus-based topologies as we faced multi-path problems, which led to the appearance of routing loops. As a consequence, we have to work further in order to extend our support to wrapped around meshes which reduce the network diameter by a factor 2. Dealing with mesh based routing, we are able to run simple examples but when the number of processes increases, we face random deadlocks which are hard to diagnose. Are they caused by MPI, our routing algorithm or VMPI_Streams ? We have to perform a survey of possible deadlocks in order to prevent such random faults which for the moment prevented us to stabilise DART for performance metric analysis purposes. We hope that the redesign of the communication layer, for MPC will fix those issues. Moreover, we plan to benchmark the use of high order meshes (more than 3 dimensions) in purpose of limiting the number of hops between blackboards while keeping a controlled amount of neighbours.

11.6 Summary

This chapter introduced the Distributed Analysis and Reduction Tree (DART) which is inspired from the distributed Blackboard paradigm to implement distributed data-flow engine. It has been designed to provide our analysis with an efficient parallelism model, able to reduce a constant data-stream. We have presented both a compact interface and reduction capabilities which can be used to implement distributed analysis. Then, we illustrated possible uses of this method with two analysis examples. Eventually, we detailed current limitations of this early implementation which suffers from some design problems which might be solved when porting DART to MPC. However, in the light of these early experiments, we believe that this programming paradigm provides the flexibility and performance required to perform distributed data-flow analysis, motivating further experimentation to propel our performance analysis framework.

CHAPTER 12

Analysis

This chapter presents the various analysis which can be done thanks to our instrumentation framework in terms of debugging, validation and profiling. After introducing our test programs, we start by describing our trace based debugger which takes advantage of our crash-dump mechanism. Then we present our deadlock detection algorithms which can perform a hierarchical deadlock detection which mixes both MPI and Pthread events. Then, we develop the profiling aspect, first by introducing our reporting infrastructure, before detailing different analysis.

12.1 Tested Programs

Our trace analysis and debugging tools have been tested on a wide range of MPI applications ranging from the simple MPI benchmark to large production-grade simulation codes. We will focus on a representative subset of these programs which have been retained for simplicity and conciseness. Programs which are both profiled and debugged in the rest of this chapter.

Our first application is EulerMHD [Wol11, WJIG11] which is a middle sized C++ MPI application which simulates Euler ideal magneto-hydrodynamic at high order on a 2D Cartesian mesh. This code relies on a scalable communication scheme as it communicates on a 4-neighbour torus. It also performs a reduction (`MPI_Allreduce`) at each time-step in order to compute the global error which is used to compute following time-step's duration. This code is scalable and has been used at the scale of the Tera 100 machine (80 000 cores).

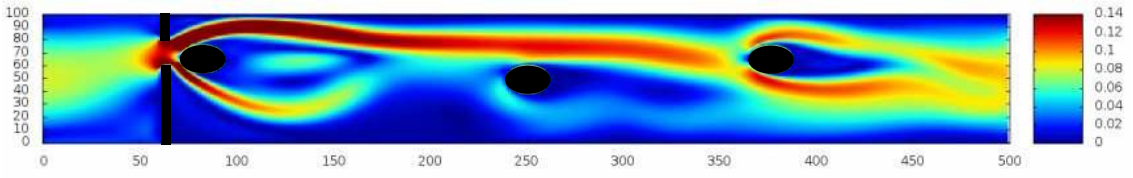


Figure 12.1: *Sample output of the lbm program.*

A second middle sized application is lbm (Lattice BoltzMann) which is a C program which simulates a Karman vortex street (see Figure 12.1 for a sample output). It has been programmed to teach program optimisation to our *Master Informatique Haute Performance et Simulation* (MIHPS) students. Consequently, it has been developed in two version, one which

features several misconceptions (propagating wait-states, centralised IOs, redundant messages, ...) and another which on the contrary has been optimised. The availability of those two versions will be interesting to illustrate bad performance patterns which can be captured by our profiling tool.

We also tested our profiling infrastructure on Hera [Jou05] which is a large C++ Adaptive Mesh Refinement (AMR) simulation platform. This application is an important test for our tools as it outlines how a complex simulation program behaves. One of the main challenge with this program is the verbosity of C++ traces which because of getters ans setters, tends to generate larger traces with important overheads (getters/setters are short functions which duration generally comparable with the instrumentation cost).

Eventually, we also tested our profiling chain on the NAS MPI Benchmark [BBB+91] which are commonly used to qualify profiling tools overhead as they solve realistic problems. They are mostly developed in Fortran and for some of them in C. Those benchmarks are available in various classes which describe the size of the simulated problem (S for small, A a bit larger and so on ...). We made our measurement using class D which allows strong scaling to several thousands of processes for most benchmarks (some have fixed limitations in terms of number of cores).

12.2 Trace-Based Debugger

Thanks to the debug buffer mechanism we introduced in section 9.4.4, out instrumentation framework can be used to explore a temporal slice which represents the last N events before program crash. This section introduces our trace-based debugger and presents the analysis its provides in term of both interactive debugging and validation.

12.2.1 Architecture

The trace-based debugger is built upon the MPC Trace reader we introduced in section 9.5. It replays the crash-dump trace in parallel while collecting program states in an exploitable fashion. In this purpose, each stream stack is replayed from individual entry and exit points and contextualised with MPI and locking events which are described as *transactions* (see section 12.2.3).

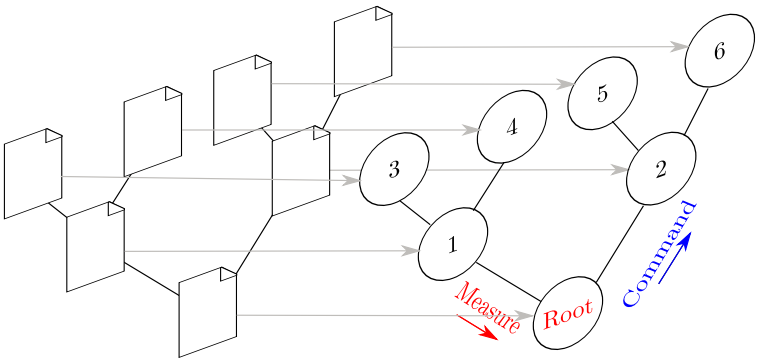


Figure 12.2: *Architecture of the trace-based debugger.*

As presented in Figure 12.2, our trace debugger relies on the parallel trace reader to distribute trace processing over several nodes. After trace processing, individual stream states are distributed among debugging processes and can be queried through a simple prompt. Commands are dispatched to individual processes and processed locally, then, suitable data are sent back to the root node which displays them. This structure is in some extent similar to the one of the DDT [All13a] debugger which relies on such tree for distributed debugging.

12.2.2 Interactive Debugging

```
9   malloc at 0x10441f040 size 72
8   malloc at 0x10441f100 size 16
7   malloc at 0x10441f1e0 size 16
6   > Parameters::SetParameters()
5       > DomainDecomposition(Parameters&)
4       < DomainDecomposition(Parameters&)
3       > Parameters::AllocateTables()
2       < Parameters::AllocateTables()
1   BEGIN MPI_ALLREDUCE with MPI_COMM_WORLD
0   Process exited badly with signal Segmentation fault (11)
```

Figure 12.3: *Example of trace-based back-trace with EulerMHD.*

As presented in figure 12.3, our trace based debugger can generate simple back-traces which include function calls, MPI and allocator calls. This back-trace is generated by replaying the stack for each stream until the program gets signaled. Signal which is also stored as a trace event by the launcher (after instrumented program interruption) as a return of the Wait system call. The trace debugger supports the following list of commands:

Command	Description
help	Display command list and associated help.
open [PATH]	Open a given trace.
read	Replay the trace-based crash dump in order to generate back-traces.
set [KEY] [VALUE]	Alter debugger configuration keys (paths to external tools).
exit	Leave the debugger.
deadlock	Perform a deadlock analysis (see Section 12.2.3).
stat	Generate a simple gnuplot-based profile.
bt [DEPTH] [TYPE]	Generates a trace-based backtrace at given depth and with various types of outputs (Stdout, textual or Graphviz).
info [TID]	Display information related to a given stream identifier.
topo [TYPE]	Generate a topology map in various formats (Stdout, textual or Graphviz).
select	Incrementally select a list of stream IDs to investigate.
filter	Removes a given stream ID from the investigated processes.
reset	Select all streams (cancels select/filter effects).
List	Lists all processes and theirs IDs in a compact fashion.

Figure 12.4: *List of commands supported by the MPC Trace debugger.*

12.2.3 Hybrid Deadlock Detection

The MPC trace debugger is able to detect hybrid deadlocks on a limited set of events: Pthread mutex locks and unlocks, MPI Collectives and MPI Blocking Point to points. To do

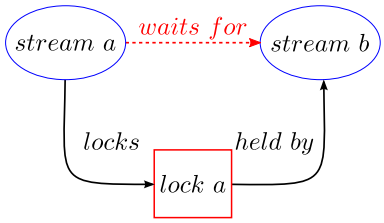
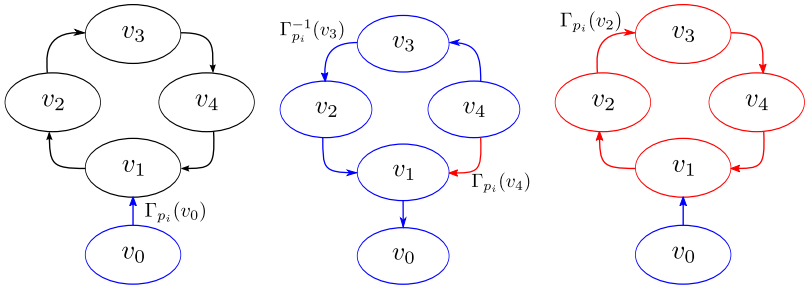


Figure 12.5: *Conversion from TRG to TWFG.*

so, it uses an abstraction called transactions. Transactions are made of two events, one when they begin and another upon their completion. We associated transactions with each of the aforementioned events. If a program deadlocks, the incriminated transaction pattern is stored into debug buffers and dumped upon program interruption. As seen in subsection 9.4.4, our instrumentation will also dump mutexes' statuses for each process. Deadlock detection uses this dump to build a dependency graph between streams, replacing each reference to a mutex address by one to the trace-id holding it (see Figure 12.5). After this process, a Task Wait For Graph (TWFG) has been generated from the Task Resource Graph (TRG). As the out degree of vertices in this graph is at most one because a stream can only make one call at a time, the Single-Resource model [SS91, Kna87] associates deadlocks with cycles in the TWFG. The same deadlock detection algorithm is applied at a each hierarchy level, propagating deadlocked states from lower levels.



(a) Propagation (b) Cycle detection (c) Labeling

Figure 12.6: *The three phases of cycle detection. Black is for* unknown *states, blue* waiting *states and red* deadlocked *states.*

Cycle Detection

deadlock detection is done in two passes of the same cycle detection algorithm which uses a simple coloration approach. The algorithm is implemented using a graph $G = (V, \Gamma)$ defined by a set of vertices V associated with a stream and Γ a function linking each vertex with its successor such as each couple generated by Γ matches a transaction. As shown by Figure 12.6, cycle detection is done by exploring the path $p_i(V_i, \Gamma_{p_i})$ from the first uncolored node while generating $\Gamma^{-1}_{p_i}$ which is the predecessor function for path p_i (see figure 12.6(a)). If for a node of the path p_i, $\Gamma_{p_i}(v_i)$ evaluates to a node v_{i+1} such as $C(v_i) = C(v_{i+1})$, then a cycle has been detected (Figure 12.6(b)) this causes the algorithms to walk back the path using $\Gamma^{-1}_{p_i}$ while flagging the nodes as deadlocked until v_{i+1}. The resulting graph is shown in figure 12.6(c).

Hierarchical Components Labeling

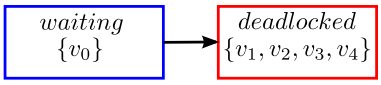
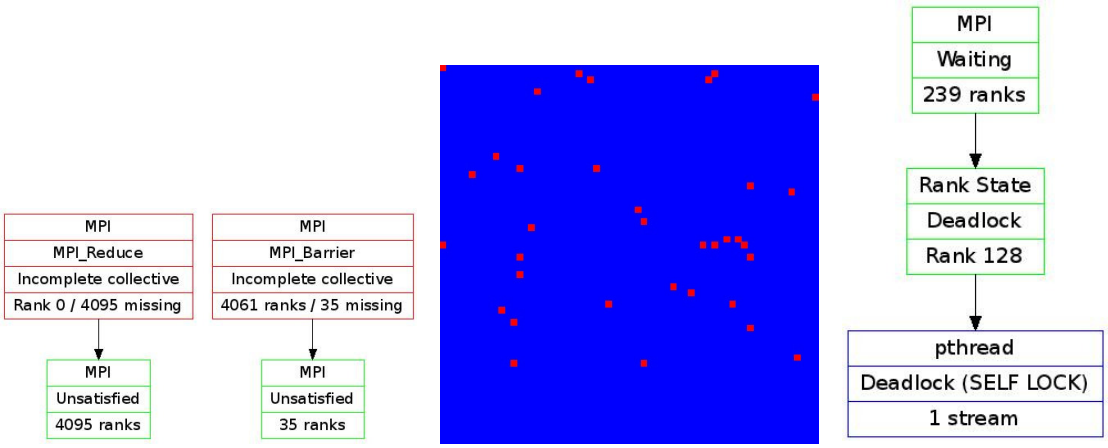


Figure 12.7: *Set-based graph associated with the cycle detection graph of Figure 12.6(c).*

Each connected component of the cycle detection graph can be seen as a graph representing the relationship between groups of processes (see Figure 12.7). To do so, for each path, vertices sharing the same state and consecutive in the cycle detection graph are grouped into sets. After a first pass on Pthread calls, MPI deadlock detection is done by merging the states of each rank's streams to a rank state. Cycle detection is then applied to propagate Pthread states through MPI point to point calls. This hierarchical component labelling greatly simplifies inconsistent states representation by grouping processes instead of displaying individual states which are not appreciable at large scale.

Experimental Results

Figure 12.8 shows outputs from the MPC trace debugger deadlock detection module on two simple test cases. Our first test consists in calling MPI_Reduce in the root process and MPI_Barrier in a random fashion among the rest of processes, the resulting pattern is given in Figure 12.8(a) while Figure 12.8(b) shows the ranks missing in `MPI_Barrier`. The second test consists in deadlocking one rank over a simple MPI ring causing its interruption. Figure 12.8(c) shows that 239 ranks among the 240 are waiting through MPI calls for rank 128 which is deadlocked.

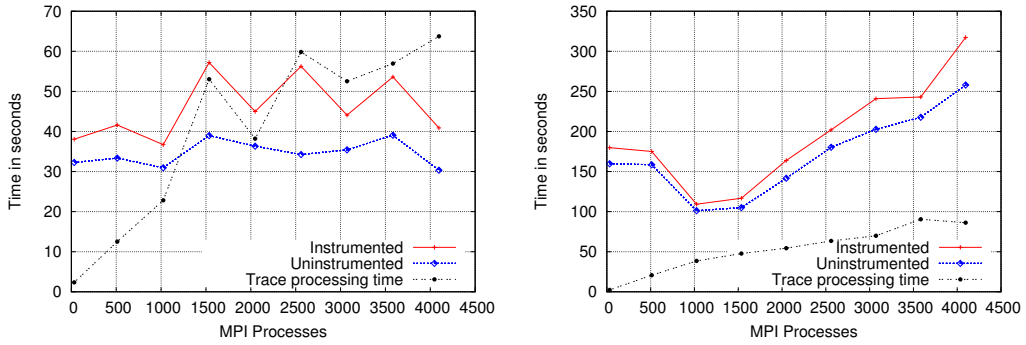


(a) Concurrent collectives (b) `MPI_Barrier` missing Ranks. (c) Deadlock on a ring

Figure 12.8: *Output of the MPC trace debugger on two deadlock cases.*

12.2.4 Trace-Based Crash-Dumps Performance

As presented in Figure 12.9, trace-based crash-dumps have been tested on Tera 100 on both EulerMHD [MWIG11] and Hera [Jou05]. shows instrumented and uninstrumented wall clock times for both applications. Instrumentation of MPI calls, mutex locks and mutex unlocks is done using debug buffers of 1024 events per process which are flushed upon completion. In order to keep computing time in a reduced range, we increased the problem size with the number of processes. The maximum overhead is close to 20%, it accounts for instrumentation dilation, time-stamps synchronisation and trace buffer flush to disk. In both cases, trace size increases linearly from 4 MB at 32 processes to 512 MB at 4096 processes. Figure 12.9 also indicates the time needed to process each crash dump. Processing has been done with an empty handler called for every trace event. Consequently, trace processing time accounts for the time needed to load and dispatch context information, plus the time to read in parallel all the events. This time increases in both cases in a close to linear fashion, showing the scalability of our hierarchical trace format and parallel trace reader. This increasing overhead mainly comes from file listing which is performed sequentially by rank 0 in order to dispatch event files at the beginning of the trace analysis.



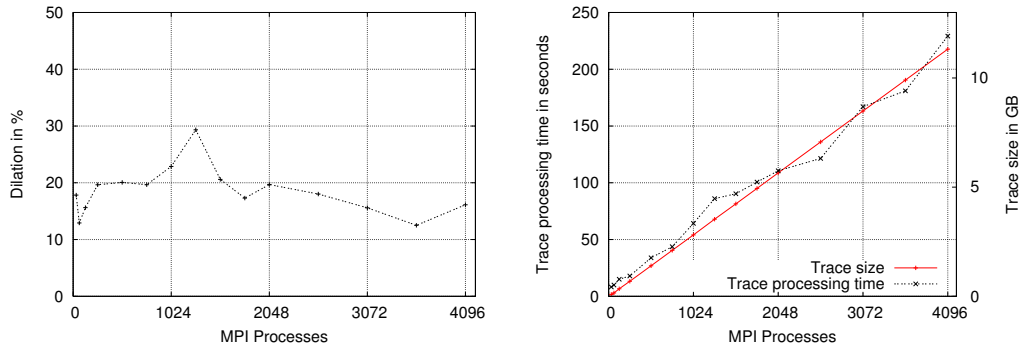
(a) EulerMHD (b) Hera

Figure 12.9: *Wall-times of instrumented and uninstrumented executions for EulerMHD and Hera on increasing problem sizes with debug buffers of 1024 events.*

12.2.5 Trace-Based Crash-Dumps and Profiling

It is also possible to rely on debug buffers to do profiling traces. In this mode, IO are avoided by storing events in a FIFO buffer. Using this method, profiling traces are less intrusive as trace buffers do not have to be flushed anymore, limiting perturbation. However, if debug buffers happen to be full, only an event subset (last N events) is available. Buffers are flushed upon program completion by the launcher wrapper, thus, the instrumented program does not perform IOs at all. Then, thanks to the unified trace reader, the trace can be processed indifferently by either our profiling tool or the trace-based debugger.

Figure 12.10 shows performance results from the instrumentation of MPI plus memory allocator on EulerMHD with debug buffers of 100000 events per process on the Curie supercomputer. Dilations are calculated from wall-times including time-stamps synchronisation



(a) EulerMHD: instrumentation.

(b) EulerMHD: trace processing.

Figure 12.10: *Instrumentation dilation (MPI and memory allocator) and trace processing performance for EulerMHD with debug buffers of 100000 events per process.*

and debug buffers flush. As shown in figure 12.10(a) overhead remains close to 20% at any scale demonstrating the scalability of our instrumentation. Figure 12.10(b) presents the time needed to generate profiling reports with the MPC trace analyser for each crash dump size. For a trace size increasing linearly from 80 MB to 11.32 GB, the time needed to generate a report in parallel on the same number of cores increases also linearly from 8.2 seconds at 32 processes to 229.19 seconds for 4096 processes. The linear increase despite the increasing number of cores (which then process a constant amount of data) comes from two main reasons, (1) sequential trace processing performed by the root process before launching the distributed analysis. Indeed, it has to scan recursively trace sub-directories in order to generate identifiers, process which takes more time at larger scales. Moreover, (2) analysis we performed during report rendering can have a cost which depends from the overall number of cores (for example density maps are larger), yielding an increasing cost as the rendering is also done sequentially.

12.3 Reporting

On the trace analyser side, in both trace-based and on-line cases, reports are represented as a tree which matches the document layout. It is therefore possible to generate a document containing sections describing various analysis. This report tree is present in each analysis node in order to perform a distributed trace reading before performing a reduction which will produce the final metrics which are rendered by the root process. Consequently, the trace analyser relies extensively on the MPC Trace reader (in its first implementation) to read the trace and parse meta-data — analysis modules being registered to individual or multiple events. In this purpose, as presented in Figure 12.11, modules called *measure collectors* are gathered in a tree and registered to valuable events. Then, the trace is processed in parallel with events reduced by individual modules. Followingly, once the whole trace has been processed a reduction is performed between *collectors* in order to gather all the data in the root process which can then perform the report rendering relying on the tree structure of the report engine. Moreover, thanks to this generic report representation different type of output are proposed: HTML, latex, markdown.

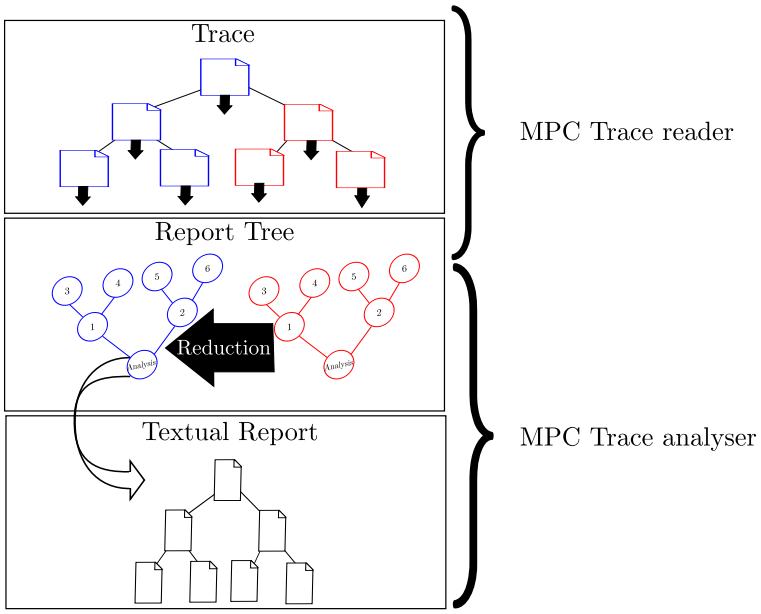


Figure 12.11: *Overview of the reporting infrastructure in the trace analyser.*

12.3.1 Measure Collectors

Function	Description
Measure_Collector_init	Initialises a measure collector for a given partition_id and event type.
Measure_Collector_release	Releases a measure collector.
Measure_Collector_set_handlers	Registers measure collector handlers (rendering, reduction, ...).
Measure_Collector_set_handlers	Registers measure collector handlers (rendering, reduction, ...).

Figure 12.12: *Measure collector interface.*

Measure collectors are the basic block of our profiling report as they convey the report structure and take care of the integration with the MPC Trace reader (blackboard in the on-line version) in order to collect performance events. As presented in Figure 12.12, measures collectors have a very compact interface as they can be registered on an event type (or on every event types). Moreover, handlers can be defined to manage event collection, reduction and rendering:

- **Setup:** this handler is called when the module is loaded. It is generally used to allocate local arrays or initialise context information.
- **Unset:** this handler is called at the very end of the analysis, it is used to free arrays allocated in the setup call.
- **Reduce:** this handler is called once the whole trace has been read (or when all instrumented applications have ended in the on-line case). It is used to reduce collected data in the root report in purpose of performing the rendering.

- **Push:** this handler is called in a thread-safe fashion for every events matching the types provided at collector initialisation. It is the main entry point to reduce data in the analyser.

Each handler is called in depth first order (relatively to the Measure collector tree) in order to guarantee a consistent order when running on several processes (particularly for reductions). Then the rendering is called solely on the root process which generates the final report in the chosen output format. This approach simplifies the design of modules as they can be developed in a very orthogonal fashion (thanks to the compact interface of Figure 12.12). Moreover, when relying on the parallel blackboard which is used in the on-line approach those modules can be developed in separated shared libraries which can themselves be linked to other libraries (for example for digital image processing).

12.3.2 Module Example

```
/* MODULE DECLARATION */
void Sample_Analysis_init( struct Measure_Collector *mc, uint64_t partition_id ) {
  /* Register module and hook to every events MALP_EVENT_ANY
   *(either MALP_EVENT_ANY or values from MALP_Trace_event_type in Event_Desc.h )*/
  Measure_Collector_init( (void *)mc, "Sample Analysis",
                          "This is a sample analysis", partition_id, MALP_EVENT_ANY );

  /* Register Handlers */
  Measure_Collector_set_handlers( (void *)mc, Sample_Analysis_push /* PUSH*/ ,
                                  Sample_Analysis_reduce /* REDUCE */,
                                  Sample_Analysis_render /* RENDER */,
                                  Sample_Analysis_setup /* SETUP */,
                                  Sample_Analysis_unset /* RELEASE */);
}
/* MODULE HOOKS ---------- IN CALL ORDER */
void Sample_Analysis_setup( struct Measure_Collector *mc ) {
  /* Called before processing any event */
}
/* PUSH IS CALLED FOR EVERY EVENT */
void Sample_Analysis_push( struct MALP_Trace_Event *evt, struct Measure_Collector *mc ) {
  switch( evt->type ) {
    case MALP_EVENT_META:       //CONTEXT INFORMATION
    break;
    case MALP_EVENT_MPI:        //MPI EVENTS
    break;
    case MALP_EVENT_WRAPPED:    //POSIX EVENTS
    break;
  }
}

void Sample_Analysis_reduce( struct Measure_Collector *mc ) {/* Called just before rendering */}
void Sample_Analysis_render( struct Measure_Collector *mc ) {/* Called before releasing */}
void Sample_Analysis_unset( struct Measure_Collector *mc ) {/* Called at the end (no more events) */}
```

Figure 12.13: *Minimal measure collector declaration for a sample analysis, called on every events.*

Figure 12.13 presents a sample module implementation. In `Sample_Analysis_init`, various handlers are registered in order to initialise, reduce, render or push events. As aforemen-

tioned, modules are integrated in a report tree which provides several rendering options (not detailed) among which are for example drawing primitives density maps or table generation.

12.4 Profiling

This section presents profiling reports excerpts from various applications in order to illustrate our performance analysis. Most of those measurements are available either trough our trace-based approach (MPC Trace library) or were ported to MALP, therefore relying on a on-line approach. Reports are 20-70 pages latex documents which describe one or several programs (when running in MALP). In the remainder of this section we present each analysis, explaining its purpose and measurement principle while presenting sample outputs from tests programs which were described in section 12.1.

12.4.1 Profiles

Our instrumentation framework is able to generate three types of profiles:

- **MPI Profiles**: with MPI calls, transmitted size and duration.
- **POSIX Interface Profiles**: with duration and total size when suitable.
- **Program functions**: with number of hits, total time and code locus.

Operation	Hits	Time	Avg time	%	Datas	Avg Datas
mkdir	4096	2 m 42 s	39.63 ms	3	-	-
fopen	4099	2 m 18 s	33.88 ms	2.5	-	-
fgets	393216	47.24 s	120.1 us	0.86	95.62 MB	255 B
memcpy	81247392	15.88 s	195.2 ns	0.29	91.62 GB	1.18 KB
write	15545	5.519 s	355 us	0.1	1.57 MB	105 B
memset	68782153	4.51 s	65.56 ns	0.082	11.80 GB	184 B
malloc	14182495	2.508 s	176.7 ns	0.046	5.28 GB	399 B
free	6800179	727.2 ms	106.7 ns	0.013	-	-
sscanf	126976	117.5 ms	925.6 ns	0.0021	-	-
fclose	4098	51.27 ms	12.51 us	0.00093	-	-
sprintf	20525	43.29 ms	2.109 us	0.00079	-	-
strlen	533641	31.81 ms	59.26 ns	0.00058	-	-
posix_memalign	579	6.65 ms	11.48 us	0.00012	-	-
gethostname	4096	4.189 ms	1.023 us	7.6e-05	-	-
fprintf	541	813.3 us	1.503 us	1.5e-05	2.28 KB	4 B
lrand48	2121	798.8 us	376.3 ns	1.5e-05	-	-
memmove	8190	662.7 us	80.74 ns	1.2e-05	-	-
fflush	8	1.336 us	167 ns	2.4e-08	-	-

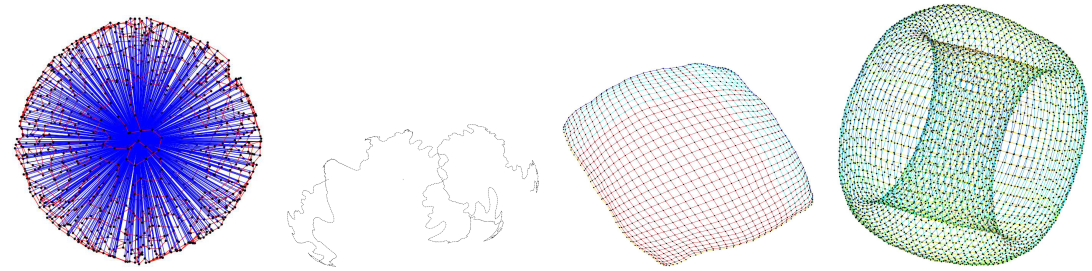
Figure 12.14: *POSIX interface profile when running EulerMHD on the Curie supercomputer with 4096 cores.*

MPI Operation	Hits	Time	Avg time	%	Datas	Avg Datas
MPI_Allreduce	544768	51 m 50 s	5.709 ms	57	4.39 MB	8 B
MPI_Wait	34111488	10 m 12 s	17.94 us	11	-	-
MPI_Isend	17055744	32.99 s	1.934 us	0.6	43.99 GB	2.70 KB
MPI_Irecv	17055744	4.237 s	248.2 ns	0.077	43.99 GB	2.70 KB
MPI_Comm_rank	4096	16.54 ms	4.039 us	0.0003	-	-
MPI_Comm_size	4096	1.423 ms	347.4 ns	2.6e-05	-	-

Figure 12.15: *MPI Profile when running EulerMHD on the Curie supercomputer with 4096 cores.*

These profiles are presented in the latex report exactly as those of figure 12.14 for POSIX function calls and 12.15 for MPI calls. Their generation relies on simple static arrays which are incremented by each events. Dealing with the function profile, we rely on a stack replay approach which matches entry and exit events in order to generate actual duration. From these profiles we can identify some performance problems, for example in Figure 12.14, we can see that the result directory is created by every processes (4096 calls to `mkdir`) — unnecessarily soliciting the file-system. On the MPI side, we can see in Figure 12.15 that EulerMHD, spends 57% of the time in `MPI_Wait`, revealing an important communication overhead, although it shall be noted that this measure were done with a reduced test case. It can been seen that EulerMHD does not recover communications (see figure 12.18(a)) — directly impeding communication costs to the application ($\approx$ 20% on a nominal test case.

12.4.2 MPI Communication Mapping

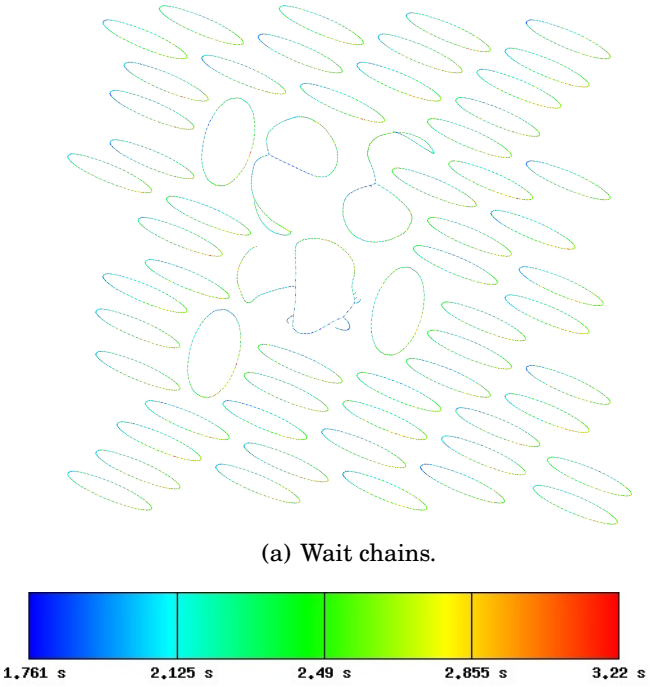


(a) lbm with IOs (1024 cores).
(b) lbm without IOs (1024 cores).
(c) NAS Lu.D (1024 cores).
(d) EulerMHD (4096 cores).

Figure 12.16: *Sample weighted communication topologies.*

Figure 12.16 presents topology maps generated by our analyser using the Graphviz tool [GN00]. Figure 12.16(a) presents the topology associated with lbm communications with IOs activated. It can be seen that the rank 0 is a central communication point, having to successively gather data-blocks from each process. Whereas, Figure 12.16(b) presents the same code with IOs deactivated, greatly simplifying the communication scheme which becomes less centralised. Figure 12.16(c) depicts the topology of NAS benchmark LU, with the LU pattern in the communication size. Eventually, Figure 12.16(d) presents EulerMHD 4-neighbour torus topology on 4096 cores, emphasising space imbalance.

12.4.3 Wait State Analysis

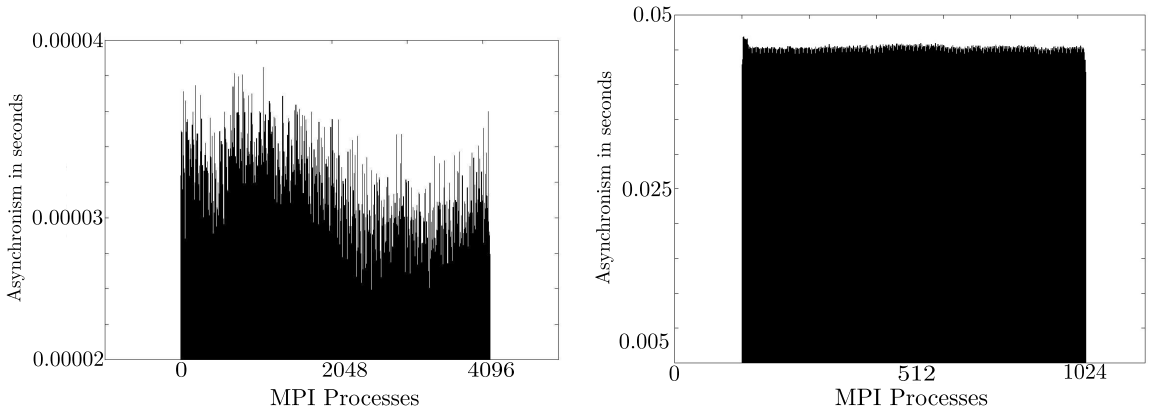


(a) Wait chains.

(b) Associated scale in seconds.

Figure 12.17: *"Most waited" topological analysis for EulerMHD on 4096 cores.*

Figure 12.17 presents the wait-graph for EulerMHD at 4096 cores on the Curie supercomputer. This directed graph is built by linking each process with the process it waits the most in terms of cumulative waiting time, thus, revealing possible dependency chains. For example, we can in if Figure 12.17 which matches the topology of Figure 12.16(d) that the torus-based topology which contains cycles tends to created circular dependencies.

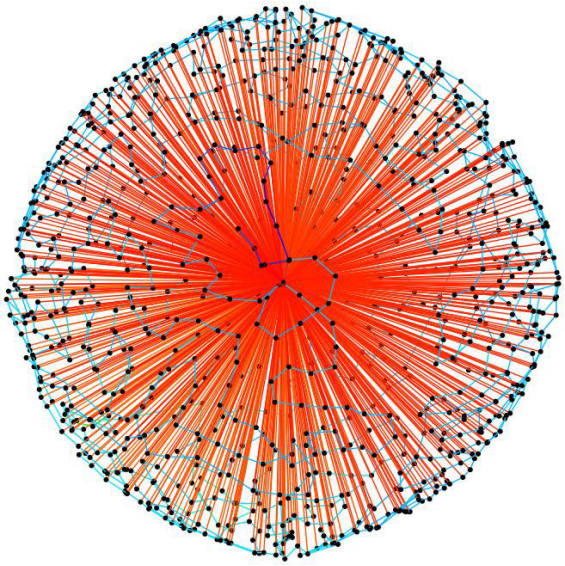


(a) EulerMHD.

(b) lbm.

Figure 12.18: *Asynchronism level comparison between EulerMHD and lbm.*

Dealing with Figure 12.18, it depicts the average anachronisms in-between communication issue and first wait, providing information over the level of asynchronism provided by the instrumented program's communications. If figure 12.18(a), witch presents the level of asynchronism for EulerMHD on 4096 processes, we can see that communications are immediately waited with an average asynchronism of $30\mu s$ — so small that it is subject to measurement noise. On the contrary, in Figure 12.18(b), we can see that lbm (with IO deactivated) has a much better asynchronism with close to $50ms$ between communication issue and first wait. This lack of asynchronism is at the origin of the dependency chains we observe in Figure 12.17 as communications are always waited for in the same order.



(a) MPI_Wait Topology.

(b) Associated scale in seconds.

Figure 12.19: *Overview of lbm topology with IOs on Curie (1024 processes).*

Figure 12.19 presents the wait topology for lmb with IOs activated. It immediately reveals the main source of inefficiency as each computing process has to wait for the $N - 2$ others which have to send their local data to the root process (see Figure 12.22(b)) before continuing to the next computation step.

12.4.4 Time Matrix

The *Time Matrix* module aims at combining both spatial and temporal aspects in a compact fashion. In this purpose, it displays metrics in a space time coordinate system thanks to a colour coding of values over a linear scale. In order to bound memory requirements, performance values are projected on a fixed size array, using a simple proportionality rule between the wall-time and sample time, each cell of this time matrix being defined by Δt and Δp respectively space and time steps. When dealing with on-line analysis, wall-times are not available,

therefore preventing Δt from being fixed — requiring a regular resampling of the matrix. Moreover, as time matrices tend to be large with an irregular repartition of samples (some events such as allocations phases are very local), we developed a sparse array with reduction support, providing two advantages: (1) gains in space (in most cases) but also (2) limiting the re-sampling to existing samples only instead of scanning the whole matrix.

Sparse Matrix Reduction

Our sparse array is based on a hash table which can be used to abstract tree based reductions. It relies on a binary tree and can be initialised with a set of functions which are sufficient to build our sparse array:

- `Red_func`: a function which takes two elements as arguments and reduces them into one another.
- `Test_func`: a function which returns true when two elements are equal (i.e. reducible to one another)
- `Key_func`: returns the key of a given element.

Using those functions our reduction hash table is able to serialise all the elements it contains before sending it to the parent node which does the following for each incoming element:

1. For each child (right and left) query the hash table for a similar element using thanks to `Key_func` and `Test_func`.
   a. If it is present operate the reduction (using `Red_func`) and push the result locally.
   b. Or simply push the element locally.
2. If not the root serialise and send to parent.

A framework which can be used to implement a sparse array sum reduce as presented in Figure 12.20. Obviously, if the matrix if full, memory consumption will be higher because of the supplementary offset parameter which is required to disambiguate in-between elements which are referred to via a key. Moreover, the hash table internally stores the key in a 64bit value, leading to a space requirement of $64 + 32 + 64 = 160$ instead of a single 64 bits values. Therefore, this size ratios defines in a straightforward way the optimality limit O_s of this method in terms of space as follows:

$$O_s = \frac{64}{160} = \frac{2}{5}$$

Consequently, this approach is advantageous only if less than $\frac{2}{5}$ of cells carry a value and leads to a threefold increase in memory consumption if the matrix if full. Naturally, not all time matrices carry sparse data, preventing this method which is currently implemented as Boolean switch in our configuration from being fully space efficient. Further development will explore the possibility of mutating the time matrix towards a static array once the space optimality limit is reached. Moreover, although it simplifies the re-sampling process by avoiding a full matrix scan, later processing such as rendering which does a systematic scan suffer

from the extra access time caused either by colliding keys in our has table or reduced cache efficiency. Cache efficiency which could be enhanced thanks to spatial hashing such as modulo which might preserve spatial locality[1] or again thanks to a data-structure mutation.

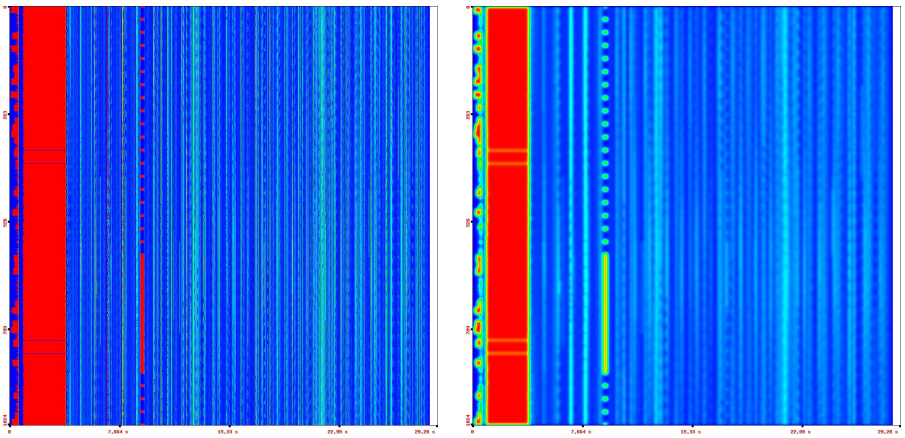
```
//Point declaration
struct Dataset_1D_point {
  uint32_t x;
  uint64_t value;
};

//Reduction function in 1D
void Dataset_1D_point_reduce(void *pa, void *pb) {
  struct Dataset_1D_point *a = (struct Dataset_1D_point *) pa;
  struct Dataset_1D_point *b = (struct Dataset_1D_point *) pb;

  a->value += b->value;
}

//Point test function for key disambiguation
int Dataset_1D_point_test(void *pa, uint64_t key, void *pb) {
  struct Dataset_1D_point *a = (struct Dataset_1D_point *) pa;
  struct Dataset_1D_point *b = (struct Dataset_1D_point *)pb;

  if( a->x == b->x )
    return 1;

  return 0;
}

//Key computation from element
uint64_t Dataset_1D_point_key(void *pa) {
  struct Dataset_1D_point *a = (struct Dataset_1D_point *) pa;

  return MALP_crc64( (char*)a, sizeof(struct Dataset_1D_point));
}
[...]
//Reduce call
Red_Ht_reduce(&dts->rht, Dataset_1D_point_reduce,
              Dataset_1D_point_test, Dataset_1D_point_key, NULL );
[...]
```

Figure 12.20: *Sparse reduction using the reduction hash table.*

Temporal and Spatial Filtering

A possible complementary step before rendering the Time Matrix is its filtering. Indeed, as presented in Figure 12.21(a), a "raw" Time Matrix is a set of discrete values with no coupling over space and time — complicating the reading. Our approach relies on a low-pass filter in order to (1) emphasise low frequency behaviours in both space and time while mitigating high frequencies, (2) convert the source discrete value set in a continuous one, (3) point out spatial and temporal correlations. In our implementation, this low-pass filter in the frequency domain in implemented using a convolution. Indeed, if we denote the convolution with $\circledast$, $F(x,y)$ and $G(x,y)$ the Fourier transforms of respectively $f(x,y)$ and $g(x,y)$ we have by definition of the *Convolution Theorem* (see Gonzalez and Woods [Gon09], section 4.2.4):

[1] Our implementation relies on our common hash table which uses CRC64 based keys.



(a) Raw data. (b) 7×7 Gaussian filtered.

Figure 12.21: *Example of Time Matrix low-pass filtering.*

$$f(x,y) \circledast g(x,y) \Leftrightarrow F(u,v) \times G(u,v) \tag{12.1}$$
$$f(x,y) \times g(x,y) \Leftrightarrow F(u,v) \circledast G(u,v) \tag{12.2}$$

Theorem which simply states the symmetry of both convolution and multiplication through the Fourier transform where one matches the other. Convolution which when dealing with digital images can be expressed in a very practical way as the multiplication of each point of a given function with all the shifted values of another one. Let us consider the following 3×3 Gaussian convolution kernel K(i,j):

K(i,j)	-1	0	1
-1	1	2	1
0	2	4	2
1	1	2	1

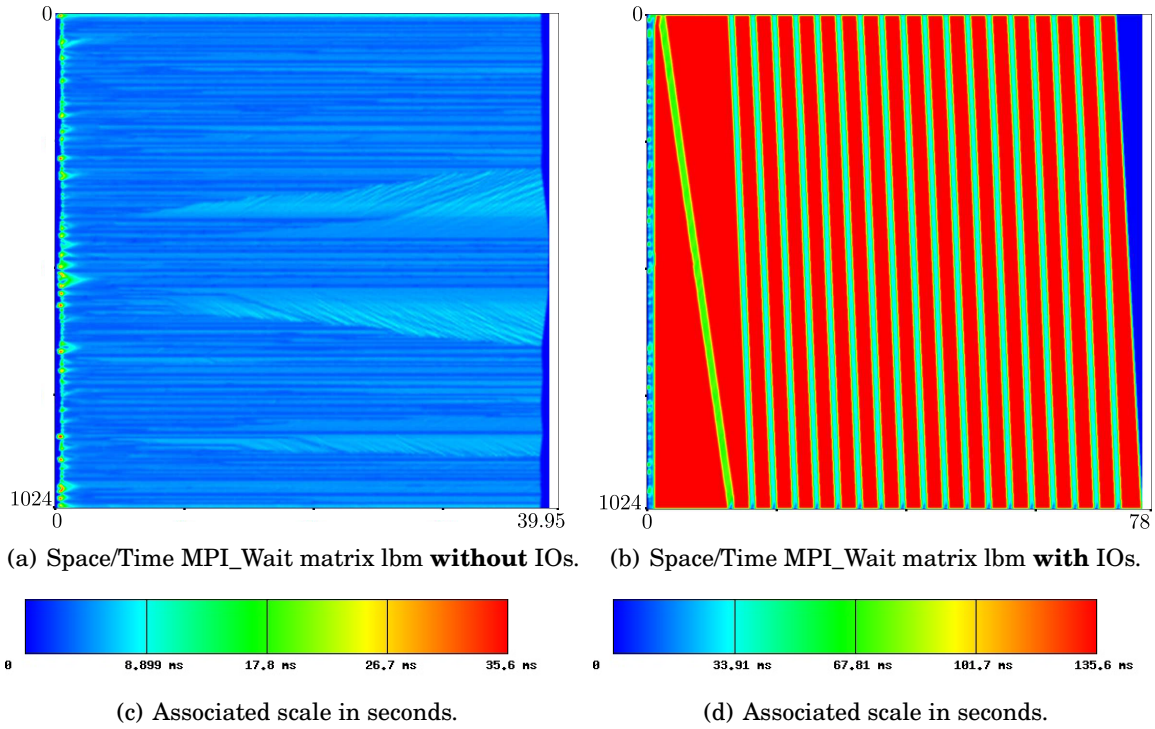
It can be applied to an image $f(x,y)$ by shifting the kernel over each point in order to generate $C_f(x,y)$ the 3×3 Gaussian filtered image of $f(x,y)$ as follows:

$$C_f(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} f(x,y)K(i,j) \tag{12.3}$$

Using this convolution equation, and the Gaussian kernel K(i,j), it is therefore possible to filter a data-set in the frequency domain without relying on an explicit transform. Moreover, thanks to the proprieties of the Gaussian function, its transform is also Gaussian with an inversely proportional bandwidth, for example, a large spatial Gaussian will be very narrow in the frequency domain, yielding a better low-pass filter. Now that we developed our simple filtering method, we can see in Figure 12.21 which presents in Figure 12.21(a) a raw version of the Time Matrix and in Figure 12.21(b) a version filtered with a 7×7 Gaussian kernel.

We can see that such filtering makes spatial and temporal dependencies more visible, simply by restoring transitions in-between states. Therefore, privileging spatially or temporally correlated states.

Sample Outputs



(a) Space/Time MPI_Wait matrix lbm **without** IOs.

(b) Space/Time MPI_Wait matrix lbm **with** IOs.

(c) Associated scale in seconds.

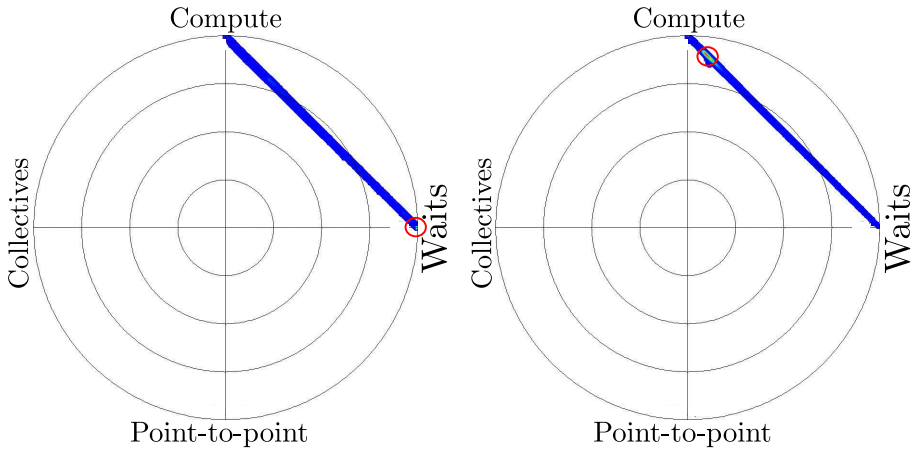
(d) Associated scale in seconds.

Figure 12.22: *"Time matrix" analysis for MPI_Wait for lmb with and without IOs when running on Curie (1024 cores).*

Figure 12.22 presents sample Time Matrix outputs on lbm either with centralised IOs activated (Figure 12.22(b)) or without (Figure 12.22(a)). It can be seen in figure 12.22(a) that represents MPI_Wait time over space (vertically) and time (horizontally) that intermediate ranks are in advance when compared to others. This phenomenon can be explained when looking a the domain decomposition (for example Figure 12.1) where simulated obstacle are centred vertically and forced to zero, requiring no computation and leading to this computational imbalance. On the contrary as shown in Figure 12.22(b) when IOs are activated, performances are catastrophic. Indeed, processes spend most of their times waiting for all processes to send their block (red areas) instead of computing (green areas). Moreover, we can also observe in figure 12.22(b), the delay caused by the communication for loop which as we can observe starts from 0 and ends at 1024, we can even see the increased cost of the first communication which accounts for queue-pair establishment.

12.4.5 MPI Quadrant

MPI Quadrants presented in Figure 12.23 are built using time-matrices in a time independent fashion. Four components: computing, collective, point to point and waiting times are gathered around the same radar chart. Then we used time matrices in order to compute the time distribution between those four components, yielding a coordinate in the MPI quadrant. This operation is repeated for each time-bucket in order to generate a density of functioning points. Using this distribution it is then possible to analyse which MPI component prevents computing (measured as non-MPI time). In Figure 12.23 we illustrate the difference between lbm with and without IOs, we can see that functioning points are forming a line between the compute and MPI waits in both cases. However, the most probable functioning point (circled in red for readability) varies from compute with a small drift towards waits when running without IOs (Figure 12.23(b)). However, when relying on centralised IOs in Figure 12.23(a) the program spends most of its time waiting with small computing phases (as seen in Figure 12.22(b)), yielding poor performance.



(a) lbm with IOs.

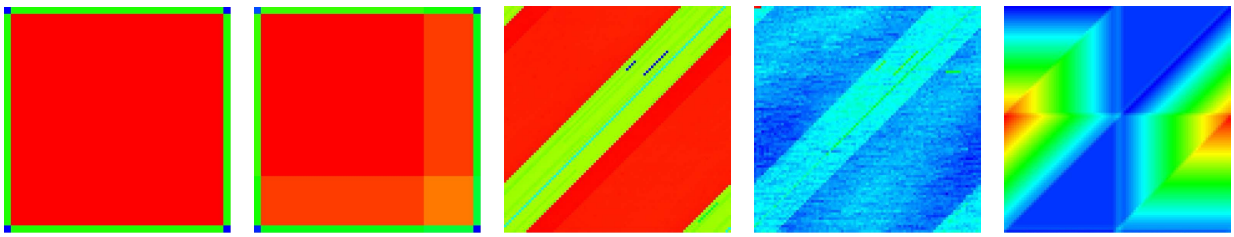
(b) lbm without IOs.

Figure 12.23: *Example of MPI Quadrants with the most probable functioning point circled in red.*

12.4.6 Spatial Analysis

Another module generates density maps for process behaviour comparison. Such maps are available for all MPI and most POSIX calls in term of hits, time and total size (when suitable) and are useful to identify spatial imbalances. For example, Figures 12.24(a) and 12.24(b) present density maps for LU.D on 1024 cores. In Figure 12.24(a), it can be clearly seen that the number of MPI_Send calls issued by the benchmark is correlated with the number of neighbours in the mesh (see Figure 12.16(c) for the associated topology). Dealing with the total size, Figure 12.24(b) shows a small imbalance which seems to follow the LU decomposition pattern. Figures 12.24(c), 12.24(d) and 12.24(e) present density maps for BT.D on 8281 cores. This example shows an imbalance in total MPI point-to-point size (see Figure 12.24(e) with blue at 660.93 MB and red at 664.87 MB). Interestingly, times spent in MPI wait calls (Figure 12.24(d)) and in collectives (Figure 12.24(c)) follow the same symmetry with nearly twice as much time between red (491.8 ms) and green areas (288.5 ms) — suggesting a possible com-

putational imbalance. Although empirical, observations made upon Figure 12.24 can provide spatial insights on codes, helping developers to understand and observe the consequences of their balancing policies.



(a) (b) (c) (d) (e)

Figure 12.24: *Sample outputs from the density map module.*

12.5 Online Trace Analysis Overhead

Our instrumentation chain has been tested on Tera 100 on NAS-MPI Benchmarks (class C and D) and EulerMHD [WJIG11]. All measurements were done three times and averaged, combinations of problem sizes and classes not supported by NAS benchmarks are omitted.

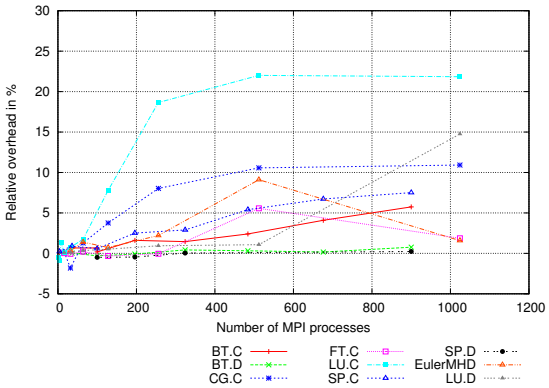


Figure 12.25: *Relative overhead for NAS Benchmarks and EulerMHD running with one analysis core for one instrumented process (on Tera 100).*

Figure 12.25 presents the relative overhead caused by our analysis tool (between MPI_Init and MPI_Finalize) when instrumenting MPI calls and their context with a $\frac{1}{1}$ ratio. This configuration maximises the bisection bandwidth and thus minimises the overhead. All overheads are all lower than 25% but vary with the application. In particular, NAS benchmarks in class C seem to expose a larger overhead than in class D. This behaviour can be understood by looking at the average instrumentation data bandwidth computed as $\overline{B_i} = \frac{\text{Total event size}}{\text{Execution time}}$. Applications with larger problem size, more time consuming in computation, issuing MPI calls less intensively, yielding a lower $\overline{B_i}$. For example, comparing SP.C and SP.D at 900 cores, we have $\overline{B_i}(\text{SP.C}) = 2.37$ GB/s and $\overline{B_i}(\text{SP.D}) = 334.99$ MB/s. Overhead is then correlated with the average instrumentation data bandwidth which has to fit in the available throughput (Figure 10.17) without impacting the instrumented program.

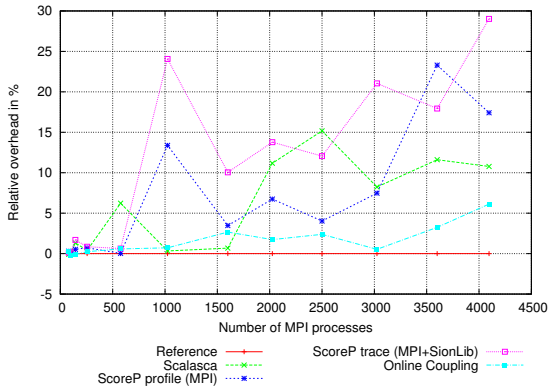


Figure 12.26: *Relative overhead with different tools for NAS Benchmarks SP.D on the Curie supercomputer (averaged 5 times).*

Figure 12.26 shows the relative overhead (between MPI_Init and MPI_Finalize) for NAS benchmark SP.D on the Curie [TOP12] supercomputer, comparing our method with two profiling tools: Scalasca 1.4.3 [SGS+11, GWW+10] and ScoreP 1.1.1 [aMBB+12]. This last one generates either OTF2 traces (compatible with Vampir [NAW+96]) or runtime profiles for Scalasca or Tau. Measurements are done with default buffer configuration for MPI only (compiler instrumentation disabled), using SionLib [FWP09] when generating ScoreP traces. It can be seen that on this benchmark, our on-line instrumentation has an overhead lower than file based traces despite manipulating larger volumes of data (ScoreP traces grow linearly from 313 MB to 116 GB and online coupling ones from 923.93 MB to 333.22 GB). This suggests that runtime-coupling is more scalable than the trace-based approach which might suffer from file-system limitations. We have no definitive explanation for the variations observed with other tools, they might be caused by varying machine load or process layouts (although nodes were allocated exclusively). Moreover, it shall be noted that NAS benchmarks have a reduced wall-time at larger scale, making them more subject to measurement noise despite several averaging passes.

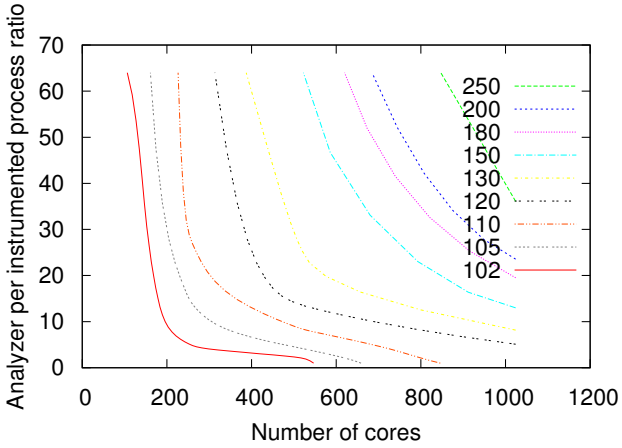


Figure 12.27: *Overhead isolines for LU.D in strong scaling.*

Figure 12.27 shows the relative overhead isolines in percents for NAS Benchmark LU.D with MPI instrumentation. As expected best performances are obtained with a ratio of $\frac{1}{1}$. Nonetheless, it is possible to operate a resource–overhead trade-off by using less analysers while keeping acceptable performances. However, at larger scales, the rapid decrease of the bisection bandwidth when lowering ratio (figure 10.17) combined with the increase of $\overline{B_i}$ because of strong scaling (from 0.47 MB/s to 2.58 GB/s) tends to reduce the "acceptable" overhead interval requiring higher ratios.

12.6 Summary

This chapter described various analysis which can be performed with our profiling and debugging tools. We firstly introduced the debugging support which was provided by our trace-based debugger with back-traces and hierarchical deadlock detection. Then, after describing our reporting interface, we pursued with the profiling part which gathers most analysis. We presented several analysis including profiles, spatial and spatio-temporal analysis. Moreover, we have shown with sample codes how such analysis can highlight performance problems. We presented outputs gathered in a structured latex report, possibly gathering analysis for several applications (when running on-line). One limitation of our analysis approach is the adherence to the rendering tools and the absence of intermediate analysis storage, preventing profiles exchange except under their report layout. We will have to overcome this limitation in following versions as if users are not likely to share traces, they might have to store raw reduced data for further processing (transverse metrics, expert consulting, ...). An example of such approach is ScoreP [aMBB+12] reports which can be opened by several tools allowing for example the invocation of Vampir from Scalasca — combining tools advantages. This stresses the fact that tools have to be included in work-flows, linking not only programs with programmers but also programmers to each other. It is therefore necessary to generate transmissible measurements, by relying on standard data-exchange technologies.

Part IV

Conclusion and Perspectives

CHAPTER 13

Conclusion

> **La conception est une action cognitive finalisée, et donc intelligente**
> C'est la quête de solutions possibles à des problèmes artificiellement posés qui guide en permanence la démarche du concepteur : elle est, par construction, tâtonnante, s'auto jalonnant d'objectifs intermédiaires, mettant en œuvre de multiples heuristiques. Elle postule qu'elle *peut avoir à connaître* des résultats qui n'existent pas encore... et que pourtant elle trouvera peut-être.
>
> J.L.L. Moigne in Intelligence et Conception [Moi86]

This thesis has introduced new developer tools which dealt with both profiling and debugging aspects. We first outlined development cycle macroscopic aspects. In Section 3.2, we insisted on the distributed nature of the development process which involves several entities (client, developer, programs) in a recursive communication scheme (structural loops) which purpose is to find a *satisficing*[1] trade-off relatively to external constraints (catalysing loops) which give context to the development effort. Process similar to the management of ill-structured problems proposed by H.A. Simon which states that once divided, a complex problem (in terms of required computation and solution space size) can be reduced to several well structured problems with the added coordination pitfall. Thus, requiring a "long-term memory" which "is literally a distributed memory, divided among the various groups of experts who are involved at one or another stage of the design process." [Sim74](Section 3.3). Memory "which continually modifies the problem space by evoking from long-term memory, new constraints, new sub-goals, and new generators for design alternatives."(*ibid* Section 3.2.3). Description which fits perfectly both development and simulation tasks, simply because they are both ill structured problems with unbounded solution spaces.

Our tools address a small aspect of this much wider context which includes both the development cycle and transitively the simulation process. Our purpose is to preserve *measure* in the complex environment of massively parallel simulation programs in order to help programmers who are facing either faults or performance problems. In this purpose we developed our contribution in part III, by firstly providing in Chapter 7 a way to understand the empirical behaviour of an MPI implementation, highlighting the presence of noise and punctual effects such as queue pair establishment which might create a certain form of performance unpredictability. Then, Chapter 8 introduced our time-synchronisation method which is a

[1] Neologism proposed by H.A. Simon [Sim97] for a solution which instead of being the optimal solutions aims at covering most needs in a *satisfying* manner.

prerequisite to any measure in a distributed environment. Then, we proceeded with the description of two successive approaches which gave frame to our performance and debugging tools: trace-based and on-line coupling.

Trace-Based approach, described in Chapter 9 introduced the MPC trace library which is a complete profiling and debugging framework. We firstly developed a new trace format which provided an advantageous alternative to OTF1[2] by simplifying meta-data management thanks to a hierarchical layout (see Section 9.4.1) while offering better compression ratios (Section 9.4.6). We also introduced our debug buffers which allowed both debugging and profiling of faulty application thanks to a shared memory buffer which is flushed upon crash (Section 9.4.4). Eventually, we presented a generic analysis tool interface which can be used to process MPC traces in parallel, greatly simplifying tool design and meta-data management. In Chapter 10, MALP an on-line approach succeeded to the MPC Trace library in order to leverage IO related limitations. In this purpose, it has been designed to completely avoid file-based traces by relying on an on-line coupling method and a parallel data-flow analysis, greatly simplifying analysis expression. We firstly introduced in Section 10.2 our coupling method which sandboxes applications in partitions, map them to each other and provides a coupling method with a behaviour close to UNIX pipes in-between partitions — completely abstracting coupling topology. In a second time, we introduced in Section 10.3 our parallel data-flow engine which is inspired from Blackboard Systems. Then, we described our multi-analysis support which can be straightforwardly implemented thanks to runtime coupling and multi-level blackboard. Eventually, we highlighted the limitations of this approach which does not furnish a data-flow based distributed communication abstraction, requiring the explicit use of synchronising MPI calls — requirement which led to the development of the Distributed Analysis and Reduction Tree (DART). Chapter 11 introduced the DART engine which relies on a network engine in order to connect several blackboard. We firstly introduced our topology management and routing policy in Section 11.2. Then Section 11.9 detailed its interface with a simple example followed by two motivating analysis. Eventually, Chapter 12 presented analysis which were developed on top of both our trace-based and on-line implementations. Firstly developing the debugging aspect in Section 12.2 before detailing our deadlock detection algorithm in Section 12.2.3. Then, we proceeded with profiling related analysis, with both temporal and spatial analysis before concluding with a performance analysis of our on-line analysis framework, demonstrating its scalability and reduced overhead.

In summary, this work developed an alternative to trace-based coupling without sacrificing all its advantages among which are modularity and event granularity. As several modules can be loaded, the on-line analysis can be as modular as a post-mortem one at the condition that enough computing power is available to process data “on-the-fly”. Dealing with granularity, not reducing events before analysing them avoids to constrain too early the type of analysis. Propriety which is guaranteed by on-line analysis thanks to a high coupling bandwidth and a pipe-lined processing which avoids storage of verbose data, as it would be done with a file-based trace possibly exhausting file-system space for a long running job (days – weeks). Then, instead of relying on a shared resource as file-system which is subject to scalability limitations, we propose to rely on the network and a distributed processing engine which takes advantage of the large number of cores in high-end systems.

[2] OTF2 was not available when this work started.

CHAPTER 14

Perspectives

Despite its has drastically evolved since its first version, MALP still suffers from several limitation. Consequently, there are still several research and development axis which require our attention in purpose of stabilising MALP's features in a robust product. This chapter proposes to list our main ideas and some basic implementation guidelines. Ideas which can be regrouped in two categories *analysis* and *features*.

14.1 Analysis

For now the set of analysis supported by MALP is clearly MPI centred despite the POSIX interface is fully instrumented. Consequently, we have to implement analysis on important aspects such as IOs or memory. Moreover, the architecture of our analysis allows "real-time" measurements which would be interesting to provide developers with a "dashboard" for their long running applications. A feature with is crucial is the support of OpenMP which is important to qualify hybrid applications in the context of upcoming supercomputer architectures. We plan to rely on the standard OpenMP instrumentation interface which is currently under specification in order to be inter-operable with most runtimes. We plan to implement various analysis among which are for example:

- An OpenMP region balancing analysis which would allow developers to see if the processing is evenly balanced over processing units. This could be implemented using a distribution reduction for each parallel region.
- A simple region profiling with the time and estimated speedup for each *parallel for* considering that the OpenMP runtime overhead can be estimated.
- An instantaneous parallelism analysis (similar to what is displayed in Vtune) which gives on a temporal axis the number of running execution streams. This would allow the identification of sequential bottlenecks.

Another type of analysis which could give some insight on performance is the phase-based profiling. Using user defined regions or simply functions boundaries, it could be possible to scatter the analysis (by duplicating them) over several code regions in order to allow the programmer to qualify them one by one. Idea which is similar to the Cube visualisation which projects MPI events over code regions. Method that we would like to adopt in order to enrich our profiling semantic relatively to code locations without requiring a complete rewrite of analysis.

Continuous profiling is also an aspect which is not covered by our tools. Indeed, if as we stated the development process is empirical and guided by trial and error, it is crucial to be able to compare two versions on the same code to attest of improvements. This could be done as in Cube through trace arithmetic or as in Tau trough a logging of different runs in Perfexplorer's database. In order to implement this functionality, we see three main requirements:

- Coupling to current program version by storing current commit identifier, ideally at compilation time (through a define altered by a compiler argument for example).
- Availability of a common storage database for profiling reports and naturally the autocoherency of profiles.
- Possibility of identifying in a unique way programs arguments and problem sizes in order to cluster solely programs running the same test case (for strong and weak scaling). This is required as some programs can run a wide range of test cases which sometimes have few in common. In this purpose we have to define an unified way or registering a test case which would yield an *unique hash* for a given test-case and an integer which describes the *problem size* (for weak scaling analysis).

If we manage to gather these informations, we would be able to characterise the speedup of an application over time, allowing the quick identification of performance regression or improvements. Moreover, this would capitalise and fingerprint applications on the long term with speedup, efficiency and measurement noise estimations.

14.2 Features

Dealing with the features, as discussed in Chapter 11, our distributed analysis implementation is not stabilised yet. We still have to work on the routing policy and improve the routing parallelism by either moving to MPC which provides a full MPI thread multiple support or directly relying on network sockets. We plan to merge the analysis and reduction network with our blackboard implementation to build what looks like an event driven distributed database which would implement the following interface mixing a key value data-store and a Blackboard using Node.js inspired event notations with 'closure functions':

- **set(key , value)** attribute value to 'key', store it in the blackboard for latter reference.
- **get(key)** retrieve data associated to 'key' null if it does not exist
- **delete(key)** delete 'key' and its associated data from the Blackboard
- **on(key, function, [id])** call 'function' when an element of type key is set or emitted. The optional 'id' argument allows a processing to be registered to a remote (in terms of process) event, therefore requiring data forwarding upon its occurrence.
- **emit(key, [data], [id])** emit an event of type 'key' which can optionally convey 'data'. Event which can be emitted locally or remotely thanks to the 'id' parameter.
- **reduce(key, function)** applies function to reduce two entries of type 'key' coming from child processes and sends the result to the parent. As in DART, it eventually produces ad data of type RED('key') which can be referred to by other processing. Moreover, it

would be interesting to dispatch the tree root over processing nodes using a consistent hashing technique in order to maximise client bandwidth. In such case “on” registration call over data which are the product of a reduction would be transparently mapped to the ’id’ which is the root of the reduction.

Once we will have managed to implement this new version of DART we believe will have all the necessary features to perform distributed analysis with a simpler and more compact semantic. Moreover, the rendering engine will be able to register itself to valued data (product of reduction) in order to propose a real time rendering thanks to the blackboard approach.

The current report layout is a static linear PDF files which embeds all the analysis, each in its dedicated chapter. Thanks to user feedback we have seen that PDF reports, despite being easily portable by nature do not offer much rendering and interaction possibilities. Consequently, we have a prototype of HTML5 based implementation which relies on a Node.js server which is coupled to a key value data-store embedded in a C application through a TCP socket. This architecture is promising as it offers full interactivity, including bidirectional interactions in-between the application and the analysis. Moreover, it completely separates analysis rendering from the data reduction, aspects which were mixed in our previous implementation. This approach can also be an opportunity for simulation codes to export more data than the common lightweight STDOUT output with for example curves, diagrams or complete data-structures. Features similar to the ones of state of the art debuggers such as DDT which is able to render the data stored in data-structures of an interrupted program.

Another aspect, if not the most important is inter-operability as there are a lot of profiling tools, each providing different analysis. Consequently, it is important to produce standard traces such as OTF2 which is standardised by the ScoreP framework as a common denominator for several prominent profiling tools. In this purpose, the engineering work to implement OTF2 support in MALP has been outsourced and is undergoing under my supervision at the moment I write these lines. This, in purpose of allowing a positive interaction with existing tools, moreover, this complementary trace support will reestablish iterative analysis which was lost due to the online aspect of our implementation.

Bibliography

[A+01] InfiniBandSM Trade Association et al. InfiniBand® Architecture Specification vol. 1. *Jun*, 19:1–131, 2001. pages 25

[AADS+07] Dorian C Arnold, Dong H Ahn, Bronis R De Supinski, Gregory L Lee, Barton P Miller, and Martin Schulz. Stack trace analysis for large scale debugging. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007. pages 61

[ABF+10] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010. pages 67

[AdSL+09] Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz. Scalable temporal order analysis for large scale debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 44:1–44:11, New York, NY, USA, 2009. ACM. pages 61, 62, 65, 75

[All13a] Allinea. Allinea DDT. `http://www.allinea.com/products/ddt/`, 2013. pages 62, 75, 155

[All13b] Allinea. Allinea MAP. `http://www.allinea.com/products/map/`, 2013. pages 75

[Amb03] M. Amblard. *Conventions & management*. Management (Bruxelles). De Boeck, 2003. pages 35

[aMBB+12] Dieter an Mey, Scott Biersdorff, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Christian Rössel, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Unified Performance Measurement System for Petascale Applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, 2012. pages 65, 74, 112, 123, 172, 173

[AMCT10] Laksono Adhianto, John Mellor-Crummey, and Nathan R Tallent. Effectively presenting call path profiles of application performance. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 179–188. IEEE, 2010. pages 67

[APM06] Dorian C Arnold, Gary D Pack, and Barton P Miller. Tree-based overlay networks for scalable applications. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006. pages 62, 75

[ATNW11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011. pages 24

[Aug11] Cédric Augonnet. *Scheduling Tasks over Multicore machines enhanced with accelerators: a Runtime System's Perspective*. PhD thesis, Université Bordeaux 1, 2011. pages 24

[Axe07] J. Axelson. *Serial port complete [electronic resource]: COM ports, USB virtual COM ports, and ports for embedded systems, second edition*. Complete Guides Series. Lakeview Research, LLC, 2007. pages 61

[BBB+91] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165. IEEE, 1991. pages 154

[BBC+08] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor and Study Lead, 2008. pages 19

[BBK+10] David A. Bader, Jonathan Berry, Simon Kahan, Richard Murphy, E. Jason Riedy, and Jeremiah Willcock. Graph 500 Benchmark 1 ("Search"). `http://www.graph500.org/Specifications.html`, October 2010. Version 1.1. pages 19

[BBMP06] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. Unraveling data race detection in the Intel Thread Checker. In *First Workshop on Software Tools for Multi-core Systems (STMCS), in conjunction with IEEE/ACM International Symposium on Code Generation and Optimization (CGO), March*, volume 26, 2006. pages 68

[BBvB+01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mallor, Ken Shwaber, and Jeff Sutherland. The Agile Manifesto. `http://www.agilemanifesto.org/`, 2001. pages 33

[BCF+12] O. Bressand, L. Colombet, A. Fontaine, G. Harel, and J.-B. Lekien. Hercule: A library of scientific

data management. In *CHOCS (Numéro 41), Revue Scientifique et Technique de la Direction des applications militaires*, pages 29–37. CEA,DAM, 2012. pages 74

[BD04] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1(1):91–99, 2004. pages 138

[BDMQO12] Maria Barreda, Manuel F Dolz, Rafael Mayo, and Enrique S Quintana-Ortı. Un Entorno de Análisis del Consumo de Aplicaciones Paralelas. 2012. pages 64

[Bec00] K. Beck. *Extreme Programming Explained.: Embrace Change*. An Alan R. Apt Book Series. Addison-Wesley, 2000. pages 34

[Bec10] Daniel Becker. *Timestamp Synchronization of Concurrent Events*. PhD thesis, RWTH Aachen University, volume 4 of IAS Series, Forschungszentrum Jülich, 2010. ISBN 978-3-89336-625-5. pages 70, 71

[BGRW13] Daniel Becker, Markus Geimer, Rolf Rabenseifner, and Felix Wolf. Extending the scope of the controlled logical clock. *Cluster Computing*, 16(1):171–189, 2013. pages 100

[BGWA10] David Bohme, Markus Geimer, Felix Wolf, and Lukas Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 90–100. IEEE, 2010. pages 65, 137

[BJK+95] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995. pages 23

[BLRW08] D. Becker, J.C. Linford, R. Rabenseifner, and F. Wolf. Replay-Based Synchronization of Timestamps in Event Traces of Massively Parallel Applications. In *Parallel Processing - Workshops, 2008. ICPP-W '08. International Conference on*, pages 212–219, 2008. pages 70, 71

[BM08] Holger Brunst and Bernd Mohr. Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with Vampir NG. In *OpenMP Shared Memory Parallel Programming*, pages 5–14. Springer, 2008. pages 64

[Bor07] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11:21, 2007. pages 74

[BPG10] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. PERISCOPE: An Online-Based Distributed Performance Analysis Tool. In Matthias S. MÃijller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 1–16. Springer Berlin Heidelberg, 2010. pages 66, 75

[BPJ13] Jean-Baptiste Besnard, Marc Pérache, and William Jalby. Event Streaming for Online Performance Measurements Reduction. *Fourth International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2013)*, 2013. pages 15, 123

[BRJ+10] Denis Barthou, Andres Charif Rubial, William Jalby, Souad Koliai, and Cédric Valensi. Performance tuning of x86 openmp codes with maqao. In *Tools for High Performance Computing 2009*, pages 95–113. Springer, 2010. pages 66

[BRW07] Daniel Becker, Rolf Rabenseifner, and Felix Wolf. Timestamp synchronization for event traces of large-scale messagepassing applications. In *In Proceedings of the 14th European PVM/MPI Conference*, pages 315–325. Springer, 2007. pages 70, 71

[BS02] Peter J Braam and Philip Schwan. Lustre: The intergalactic file system. In *Ottawa Linux Symposium*, page 50, 2002. pages 73

[BT87] L. Boltanski and L. Thévenot. *Les économies de la grandeur*. Cahiers du Centre d'études de l'emploi. Presses universitaires de France, 1987. pages 35

[Bul10] Bull. Bullx super-node 6010 support. `http://support.bull.com/ols/product/platforms/hw-extremcomp/hw-bullx-sup-node/bullx-s6010`, 2010. pages 26

[BWG12] David Bohme, Felix Wolf, and Markus Geimer. Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2538–2541. IEEE, 2012. pages 65

[BZ06] Emery D Berger and Benjamin G Zorn. DieHard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Notices*, volume 41, pages 158–168. ACM, 2006. pages 68

[C+95] TIS Committee et al. Tool interface standard (TIS) executable and linking format (ELF) specification, 1995. pages 109

[C+10] DWARF Debugging Information Format Committee et al. DWARF debugging information format version 4, 2010. pages 109, 111

[CB74] Donald D Chamberlin and Raymond F Boyce. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974. pages 74

[CB91] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, July 1991. http://dx.doi.org/10.1016/0020-0190(91)90055-M. pages 71

[CBC+05] P. Coteus, H. R. Bickford, T. M. Cipolla, P. G. Crumley, A. Gara, S. A. Hall, G. V. Kopcsay, A. P. Lanzetta, L. S. Mok, R. Rand, R. Swetz, T. Takken, P. La Rocca, C. Marroquin, P. R. Germann, and M. J. Jeanson. Packaging the Blue Gene/L supercomputer. In *IBM Journal of Research and Development*, volume 49, page 123, 2005. pages 68

[CCZ07] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. pages 23

[CD10] Kristina Chodorow and Michael Dirolf. *MongoDB: the definitive guide*. O'Reilly Media, 2010. pages 74

[CDFT12] Kevin Coulomb, Augustin Degomme, Mathieu Faverge, and François Trahay. An open-source tool-chain for performance analysis. In *Tools for High Performance Computing 2011*, pages 37–48. Springer, 2012. pages 64

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008. pages 74

[CGL08] Anthony Chan, William Gropp, and Ewing Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2):155–165, 2008. pages 63, 64, 112

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005. pages 23

[Coc04] A. Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. The agile software development series. Pearson Education, 2004. pages 33

[Con08] DSDM Consortium. DSDM Atern Handbook. http://www.dsdm.org/, 2008. pages 33

[Cor88] Daniel D Corkill. *Design alternatives for parallel and distributed blackboard systems*. Computer and Information Science, University of Massachusetts, 1988. pages 73

[Cor03] Daniel D Corkill. Collaborating software: Blackboard and multi-agent systems & the future. In *Proceedings of the International Lisp Conference*, volume 10, 2003. pages 72

[CPJ10] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC. In *IWOMP*, pages 1–14, 2010. pages 14

[CPJ11] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Thread-Local Storage Extension to Support Thread-Based MPI/OpenMP Applications. In *IWOMP*, pages 80–93, 2011. pages 14, 201

[Cri89] Flaviu Cristian. Probabilistic Clock Synchronization. *Distributed Computing*, 3(3):146–158, 1989. pages 70, 93, 99

[DB93] RogÃľrio Drummond and ÃŰzalp BabaoÄĺlu. Low-cost clock synchronization. *Distributed Computing*, 6(4):193–203, 1993. http://dx.doi.org/10.1007/BF02242707. pages 70

[DBB07] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007. pages 24

[DBC+05] Lamia Djoudi, Denis Barthou, Patrick Carribault, William Jalby, Christophe Lemuet, Jean-Thomas Acquaviva, et al. Exploring application performance: a new tool for a static/dynamic approach. In *LACSI Symposium, Santa Fe*, pages 41–49, 2005. pages 66

[DCPJ12] Sylvain Didelot, Patrick Carribault, Marc Pérache, and William Jalby. Improving MPI Communication Overlap with Collaborative Polling. In *EuroMPI*, pages 37–46, 2012. pages 14

[Def05] Defense Technical Information Center. Revised DoD Directive 5000.1 (Major System Acquisitions), 2005. pages 34

[Dep85] Department Of Defense (DOD). DOD-STD-2167 Military Standard: Defense System Software Development, 1985. pages 31

[Dep88] Department Of Defense (DOD). DOD-STD-2167A Military Standard: Defense System Software Development, 1988. pages 31

[Dep94] Department Of Defense (DOD). MIL-STD-498 - DI-IPSC-81433: Software Requirements Specification, December 1994. pages 34

[DG96] Peter Deutsch and Jean-Loup Gailly. RFC 1950–ZLIB Compressed Data Format Specification version 3.3. *IETF/IESG, May*, 1996. pages 63, 116

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. pages 75

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, volume 14, pages 205–220, 2007. pages 74

[DKdOS00] Jacques Chassin De Kergommeaux and Benhur de Oliveira Stein. Pajé: an extensible environment for visualizing multi-threaded programs executions. In *Euro-Par 2000 Parallel Processing*, pages 133–140. Springer, 2000. pages 63

[DKDS+05] Jayant Desouza, Bob Kuhn, Bronis R De Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with Intel® Message Checker. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 78–82. ACM, 2005. pages 67

[DKMN08] Jens Doleschal, Andreas Knupfer, Matthias S. Muller, and Wolfgang E. Nagel. Internal Timer Synchronization for Parallel Event Tracing. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 202–209. Springer Berlin Heidelberg, 2008. pages 71

[DLL07] Philippe Deniel, Thomas Leibovici, and Jacques-Charles Lafoucrière. GANESHA, a multi-usage with large cache NFSv4 server. In *Linux Symposium*, page 113, 2007. pages 74

[DLP03] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience, 2003. pages 15, 19

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. pages 23

[DMN99] Yves Denneulin, Jean-François Méhaut, and Raymond Namyst. Customizable thread scheduling directed by priorities. 1999. pages 23

[Don87] Jack Dongarra. The LINPACK Benchmark: An Explanation. In Elias N. Houstis, Theodore S. Papatheodorou, and Constantine D. Polychronopoulos, editors, *Supercomputing, 1st International Conference, Athens, Greece, June 8-12, 1987, Proceedings*, volume 297 of *Lecture Notes in Computer Science*, pages 456–474. Springer, 1987. pages 19

[dOSdKM10] B de Oliveira Stein, J Chassin de Kergommeaux, and G Mounié. Pajé trace file format. Technical report, Tech. rep.(March 2003), 2010. pages 63, 112

[DQZ90] K. J. Danhof, J. Quisenberry, and M. Zargham. Concurrency in blackboard systems. In *Proceedings of the 3rd international conference on Industrial and engineering applications of artificial intelligence and expert systems - Volume 1*, IEA/AIE '90, pages 109–113, New York, NY, USA, 1990. ACM. pages 73

[Dun92] Thomas H. Dunigan. Hypercube clock synchronization. *Concurrency - Practice and Experience*, 4(3):257–268, 1992. pages 70

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163, December 2002. http://doi.acm.org/10.1145/844128.844143. pages 70

[EGS06] Tarek El-Ghazawi and Lauren Smith. UPC: unified parallel C. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27. ACM, 2006. pages 23

[EL80] Lee D. Erman and Victor R. Lesser. The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12:213–253, 1980. pages 72

[EM88] R. Engelmore and T. Morgan. *Blackboard systems*. Insight series in artificial intelligence. Addison-Wesley, 1988. pages 72, 73

[Eng03] RS Engelschall. pth GNU Portable Threads. *Pth Manual, Online*, pages 1–31, 2003. pages 23

[ET79] Robert Engelmore and Allan Terry. Structure and Function of the CRYSALIS System. In *Proceedings of the 6th international joint conference on Artificial intelligence-Volume 1*, pages 250–256. Morgan Kaufmann Publishers Inc., 1979. pages 73

[EWG+11] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. Open Trace Format 2-The next generation of scalable trace formats and support libraries. In *Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium*, 2011. pages 63, 101, 112

[FG87] B FLAMANT and G GIRARD. Intelligence Service: construisez votre propre système expert. *Revue des Télécommunications*, 61(4):417–421, 1987. pages 72

[FG07] Karl Fürlinger and Michael Gerndt. Automated performance analysis using ASL performance properties. In *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 390–397. Springer, 2007. pages 67

[FGMM06] John Feo, John Gilbert, Kamesh Madduri, and Bill Mann. HPCS Scalable Synthetic Compact Applications #2 Graph Analysis, 2006. pages 19

[Fid88] Colin J. Fidge. Timestamps in message passing systems that preserve the partial ordering. In *Theoretical Computer Science*, 1988. pages 69, 71, 100

[fISZ13] TU Dresden Center for Information Services and High Performance Computing (ZIH). VampirTrace 5.14.3 User Manual, 2013. pages 70, 71

[Fit04] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, (124):72–74, 2004. pages 74

[FWP09] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. Scalable massively parallel I/O to task-local files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009. pages 63, 74, 123, 172

[GBP07] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23. ACM, 2007. pages 23

[Geo11] Lars George. *HBase: the definitive guide*. O'Reilly Media, Incorporated, 2011. pages 74

[GF07] Michael Gerndt and Karl Fürlinger. Specification and detection of performance problems with ASL. *Concurrency and Computation: Practice and Experience*, 19(11):1451–1464, 2007. pages 67

[GFB+04] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004. pages 23

[GFK05] Michael Gerndt, K Fürlinger, and E Kereku. Periscope: Advanced techniques for performance analysis. In *Proceedings of the 2005 International Conference on Parallel Computing (ParCo 2005)*, pages 15–26. Citeseer, 2005. pages 67

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003. pages 74

[GK07] Michael Gerndt and Edmond Kereku. Automatic memory access analysis with periscope. In *Computational Science–ICCS 2007*, pages 847–854. Springer, 2007. pages 67

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996. pages 23

[GN00] Emden R Gansner and Stephen C North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11):1203–1233, 2000. pages 142, 163

[Gon09] R.C. Gonzalez. *Digital Image Processing*. Pearson Education, 2009. pages 167

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the Very Large Database Conference*, pages 144–154, 1981. pages 74

[Gra01] Paul S Graham. *Logical hardware debuggers for FPGA-based systems*. PhD thesis, Brigham Young University, 2001. pages 61

[Gra03] A. Grama. *Introduction to Parallel Computing*. Pearson Education. Addison Wesley Publishing Company Incorporated, 2003. pages 26, 145

[GSS+12] Markus Geimer, Pavel Saviankou, Alexandre Strube, Zoltán Szebenyi, Felix Wolf, and Brian JN Wylie. Further improving the scalability of the Scalasca toolset. In *Applied Parallel and Scientific Computing*, pages 463–473. Springer, 2012. pages 65

[Gus88] John L Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988. pages 44

[GWW+10] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 2010. pages 65, 172

[GWWM09] Markus Geimer, Felix Wolf, Brian JN Wylie, and Bernd Mohr. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Computing*, 35(7):375–388, 2009. pages 65

[GZ83] Riccardo Gusella and Stefano Zatti. TEMPO: A Network Time Controller for a Distributed Berkeley UNIX System. Technical Report UCB/CSD-83-163, EECS Department, University of California, Berkeley, Dec 1983. pages 93

[Hai12] J.L. Hainaut. *Bases de données - 2e éd. - Concepts, utilisation et développement: Concepts, utilisation et développement*. Informatique. Dunod, 2012. pages 74

[HC99] M Haardt and M Coleman. ptrace (2), 1999. pages 61

[HC02] Jason Hill and David Culler. A wireless embedded sensor architecture for system-level optimization. Technical report, UC Berkeley Technical Report, 2002. pages 70

[Hig00] J.A. Highsmith. *Adaptive Software Development: An Evolutionary Approach to Controlling Chaotic Systems*. Dorset House Publishing, 2000. pages 33

[HKW11] Marc-André Hermanns, Sriram Krishnamoorthy, and Felix Wolf. A Scalable Replay-based Infrastructure for the Performance Analysis of One-sided Communication. In *Proc. of the 1st Intl. Workshop on High-performance Infrastructure for Scalable Tools (WHIST), Tucson, AZ, USA*, June 2011. pages 65

[HM01] Simon Huband and D McDonald. A preliminary topological debugger for MPI programs. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 422–429. IEEE, 2001. pages 61

[HM05] Kevin A Huck and Allen D Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 41. IEEE Computer Society, 2005. pages 66

[HM10] T. Halpin and T. Morgan. *Information Modeling and Relational Databases*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2010. pages 74

[HMBM05] Kevin A Huck, Allen D Malony, Robert Bell, and Alan Morris. Design and implementation of a parallel performance data management framework. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 473–482. IEEE, 2005. pages 65

[HMdS+12] Tobias Hilbrich, Matthias S. Müller, Bronis R. de Supinski, Martin Schulz, and Wolfgang E. Nagel. GTI: A Generic Tools Infrastructure for Event-Based Tools in Parallel Systems. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 1364–1375, Washington, DC, USA, 2012. IEEE Computer Society. pages 67, 75

[HMK09] Tobias Hilbrich, Matthias S Müller, and Bettina Krammer. MPI correctness checking for OpenMP/MPI applications. *International Journal of Parallel Programming*, 37(3):277–291, 2009. pages 67

[HMSM07] Kevin A Huck, Allen D Malony, Sameer Shende, and Alan Morris. Scalable, automated performance analysis with tau and perfexplorer. *Parallel Computing (ParCo), Aachen, Germany*, pages 1–8, 2007. pages 66

[Hoo96] Robert Hood. The p2d2 project: building a portable distributed debugger. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 127–136. ACM, 1996. pages 62

[HS02] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Syst. J.*, 41(1):4–12, January 2002. http://dx.doi.org/10.1147/sj.411.0004. pages 48

[HSC+08] Oscar Hernandez, Fengguang Song, Barbara Chapman, Jack Dongarra, Bernd Mohr, Shirley Moore, and Felix Wolf. Performance instrumentation and compiler optimizations for MPI/OpenMP applications. In *OpenMP Shared Memory Parallel Programming*, pages 267–278. Springer, 2008. pages 67

[HT99] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Pearson Education, 1999. pages 29, 49

[HWM02] Wei Huang, Zhe Wang, and Jie Ma. Design of DMPI on DAWNING-3000. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 314–322. Springer, 2002. pages 139

[IAB95] IABG. V-Model, Lifecycle Process Model. http://v-modell.iabg.de/, 1995. pages 32

[INT10] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2, 2010. pages 70

[Int12a] Intel. Intel Debugger for Linux (IDB). http://software.intel.com/en-us/articles/idb-linux/, 2012. pages 61, 62

[Int12b] Intel. Intel Trace Collector Reference Guide, 2012. pages 70, 71

[JBM12] Emily R Jacobson, Michael J Brim, and Barton P Miller. A lightweight library for building scalable tools. In *Applied Parallel and Scientific Computing*, pages 419–429. Springer, 2012. pages 65, 75

[JBR12] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Object Technology Series. Pearson Education, Limited, 2012. pages 34

[JDA+09] Heike Jagode, Jack Dongarra, Sadaf Alam, Jeffrey Vetter, Wyatt Spear, and Allen D Malony. A holistic approach for performance measurement and analysis for petascale applications. In *Computational Science–ICCS 2009*, pages 686–695. Springer, 2009. pages 65

[Jéz89] Jean-Marc Jézéquel. Building a Global Time on Parallel Machines. In *WDAG*, pages 136–147, 1989. pages 70

[Jou05] Hervé Jourdren. HERA: A Hydrodynamic AMR Platform for Multi-Physics Simulations. In *Adaptive Mesh Refinement - Theory and Applications*, volume 41 of *Lecture Notes in Computational Science and Engineering*. Springer Berlin Heidelberg, 2005. pages 154, 158

[JT08] Ali Jannesari and Walter F Tichy. On-the-fly race detection in multi-threaded programs. In *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*, page 6. ACM, 2008. pages 68

[KBB+06] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E Nagel. Introducing the open trace format (OTF). In *Computational Science–ICCS 2006*, pages 526–533. Springer, 2006. pages 63, 101, 112

[KBD+08] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The vampir performance analysis toolset. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008. pages 63, 64

[KBMS06] Holger Brunst Andreas Knüpfer, Holger Brunst, Allen D Malony, and Sameer S Shende. Open trace format api specification version 1.1. *Center for High Performance Computing University of Dresden, Germany*, 2006. pages 63

[KH06] Elliott D Kaplan and Christopher J Hegarty. *Understanding GPS: principles and applications*. Artech House Publishers, 2006. pages 71

[KHL+07] Bettina Krammer, Valentin Himmler, David Lecomber, et al. Coupling DDT and Marmot for debugging of MPI applications. *Proc. of ParCo 2007*, pages 4–7, 2007. pages 67

[KJP08] Matthew J Koop, Terry Jones, and Dhabaleswar K Panda. Mvapich-aptus: Scalable high-performance multi-transport MPI over infiniband. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008. pages 23

[KK93] Laxmikant V Kale and Sanjeev Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993. pages 23

[Kle05] Andi Kleen. Update TSC sync algorithm, Linux Kernel (commit dda50e716dc9451f40eebfb2902c260e4f62cf34), May 2005. pages 70

[KLW+03] Sushmitha P Kini, Jiuxing Liu, Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K Panda. Fast and scalable barrier using rdma and multicast mechanisms for infiniband-based clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 369–378. Springer, 2003. pages 83

[KMR04a] Bettina Krammer, Matthias S. Müller, and Michael M. Resch. MPI Application Development Using the Analysis Tool MARMOT. In *In ICCS 2004, volume LNCS 3038*. Springer, 2004. pages 67

[KMR04b] Bettina Krammer, Matthias S Müller, and Michael M Resch. MPI I/O analysis and error detection with MARMOT. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 242–250. Springer, 2004. pages 67

[Kna87] Edgar Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys (CSUR)*, 19(4):303–328, 1987. pages 156

[Kos10] Joseph Koshy. libelf by Example. *Web site: http://people.freebsd.org/jkoshy/download/libelf/article.html*, 2010. pages 109

[KP07] M Kerrisk and J Pryzby. backtrace (3), 2007. pages 61

[KR06] Bettina Krammer and Michael M Resch. Correctness checking of MPI one-sided communication using MARMOT. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 105–114. Springer, 2006. pages 67

[KZO+10] Souad Koliai, Stéphane Zuckerman, Emmanuel Oseret, Mickaël Ivascot, Tipp Moseley, Dinh Quang, and William Jalby. A balanced approach to application performance tuning. In *Languages and Compilers for Parallel Computing*, pages 111–125. Springer, 2010. pages 66

[Lab10] Jesus Labarta. StarSS: A programming model for the multicore era. In *PRACE WorkshopâĂŹNew Languages & Future Technology PrototypesâĂŹ at the Leibniz Supercomputing Centre in Garching (Germany)*, 2010. pages 24

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. http://doi.acm.org/10.1145/359545.359563. pages 69, 71

[LBFL80] Robert K Lindsay, Bruce G Buchanan, Edward A Feigenbaum, and Joshua Lederberg. Applications of artificial intelligence for organic chemistry: The DENDRAL project. *Structure*, 2(2.7):2–8, 1980. pages 72

[LC83] Victor R Lesser and Daniel G Corkill. The distributed vehicle monitoring testbed: A tool for investigating distributed problem solving networks. *AI magazine*, 4(3):15, 1983. pages 73

[Lei85] C.E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *Computers, IEEE Transactions on*, C-34(10):892–901, 1985. pages 26

[Lew69] D. Lewis. *Convention: A Philosophical Study*. Wiley(2008), 1969. pages 35

[LG98] Delon Levi and Steven A Guccione. BoardScope: A debug tool for reconfigurable systems. *Configurable Computing Technology and its uses in High Performance Computing, DSP and Systems Engineering*, pages 239–246, 1998. pages 61

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989. pages 138

[Lin90] Mark A Linton. The evolution of Dbx. In *Proceedings of the Summer USENIX Conference*, pages 211–220. Citeseer, 1990. pages 61, 62

[LM09] Avinash Lakshman and Prashant Malik. Cassandra: a structured storage system on a P2P network. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 47–47, New York, NY, USA, 2009. ACM. pages 74

[LMC99] Cheng Liao, Margaret Martonosi, and Douglas W. Clark. Experience with an adaptive globally-synchronizing clock algorithm. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '99, pages 106–114, New York, NY, USA, 1999. ACM. pages 70

[LMM11] Chee Wai Lee, Allen D Malony, and Alan Morris. TAUmon: scalable online performance data analysis in TAU. In *Euro-Par 2010 Parallel Processing Workshops*, pages 493–499. Springer, 2011. pages 66

[LPSW12] Daniel Lorenz, Peter Philippen, Dirk Schmidl, and Felix Wolf. Profiling of OpenMP tasks with Score-P. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 444–453. IEEE, 2012. pages 65

[lt11] The libunwind team. The libunwind project. http://www.nongnu.org/libunwind/, 2011. pages 61

[M+09] Aaftab Munshi et al. The opencl specification. *Khronos OpenCL Working Group*, 1:l1–15, 2009. pages 24

[Mar91] J. Martin. *Rapid Application Development*. The James Martin productivity series. MacMillan, 1991. pages 33

[Mar99] Brian Marick. New Models for Test Development. http://www.exampler.com/testing-com/writings/new-models.pdf, 1999. pages 31

[Mat88] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. 1988. pages 69, 71

[MB93] John May and Francine Berman. Panorama: A portable, extensible parallel debugger. In *ACM Sigplan Notices*, volume 28, pages 96–106. ACM, 1993. pages 62

[MBDH99] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proc. Department of Defense HPCMP Users Group Conference*, 1999. pages 64

[MBS+11] Allen D Malony, Scott Biersdorff, Sameer Shende, Heike Jagode, Stanimire Tomov, Guido Juckeland, Robert Dietrich, Duncan Poole, and Christopher Lamb. Parallel performance measurement of heterogeneous parallel systems with GPUs. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 176–185. IEEE, 2011. pages 65

[MCC+95] Barton P. Miller, Mark D. Callaghan, Jonathan M Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995. pages 65, 75

[Mey97] B. Meyer. *Object-oriented software construction*. Prentice-Hall International Series in Computer Science. Prentice Hall PTR, 1997. pages 49

[MF08] MPI-Forum. MPI: A message passing interface standard, version 2.1. 2008. pages 23, 74, 103

[MGN10] Stephanie Moreaud, Brice Goglin, and Raymond Namyst. Adaptive MPI Multirail Tuning for Non-uniform Input/Output Access. In *EuroMPI*, pages 239–248, 2010. pages 25

[MHC94] Barton P Miller, Jeffrey K Hollingsworth, and Mark D Callaghan. The paradyn parallel performance tools and pvm. 1994. pages 64

[Mil91] David L. Mills. Internet Time Synchronization: the Network Time Protocol. *IEEE Transactions on Communications*, 39:1482–1493, 1991. pages 70, 93

[MMSH10] Alan Morris, Allen D. Malony, Sameer Shende, and Kevin Huck. Design and Implementation of a Hybrid Parallel Performance Measurement System. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, 2010. pages 65

[MMSW01] Bernd Mohr, Allen D Malony, Sameer Shende, and Felix Wolf. *Towards a performance tool interface for OpenMP: An approach based on directive rewriting*. Citeseer, 2001. pages 65

[MMSW02] Bernd Mohr, Allen D Malony, Sameer Shende, and Felix Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23(1):105–128, 2002. pages 64

[Moi86] J.L.L. Moigne. *Intelligence et Conception*. Nouvelle encyclopédie des sciences et des techniques. Fondation Diderot, 1986. http://www.intelligence-complexite.org/fileadmin/docs/1306jlm86.pdf. pages 177

[Moi99] J.L.L. Moigne. *La Modélisation des systèmes complexes*. Sciences des organisations. Dunod, 1999. pages 41

[Mol07] Ingo Molnar. x86: rewrite SMP TSC sync code, Linux Kernel (commit 95492e4646e5de8b43d9a7908d6177fb737b61f0), February 2007. pages 70

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. 38(8), April 1965. pages 20

[Mor11] Stéphanie Moreaud. Mouvement de données et placement des tâches pour les communications haute performance sur machines hiérarchiques. http://tel.archives-ouvertes.fr/tel-00635651/fr/, October 2011. pages 25, 83

[MR91] Allen D. Malony and Daniel A. Reed. Models for Performance Perturbation Analysis. In *Workshop on Parallel and Distributed Debugging*, pages 15–25, 1991. pages 71

[MRW92] A.D. Malony, D.A. Reed, and Harry A G Wijshoff. Performance measurement intrusion and perturbation analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 3(4):433–450, 1992. pages 71

[MSDS93] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top500(www.top500.org). http://www.top500.org, 1993. pages 15, 19, 20

[MSM05] Allen D Malony, Sameer S Shende, and Alan Morris. Phase-based parallel performance profiling. In *Proceedings of the PARCO 2005 conference*, 2005. pages 66

[MSMS08] Alan Morris, Wyatt Spear, Allen D Malony, and Sameer Shende. Observing performance dynamics using parallel profile snapshots. In *Euro-Par 2008–Parallel Processing*, pages 162–171. Springer, 2008. pages 66

[MT95] Eric Maillet and Cécile Tron. On efficiently implementing global time for performance evaluation on multiprocessor systems. *J. Parallel Distrib. Comput.*, 28(1):84–93, July 1995. http://dx.doi.org/10.1006/jpdc.1995.1090. pages 71

[MW03] Bernd Mohr and Felix Wolf. KOJAK–A tool set for automatic performance analysis of parallel programs. In *Euro-Par 2003 Parallel Processing*, pages 1301–1304. Springer, 2003. pages 64

[MW07] Arndt Mühlenfeld and Franz Wotawa. Fault detection in multi-threaded C++ server applications. *Electronic Notes in Theoretical Computer Science*, 174(9):5–22, 2007. pages 68

[MWIG11] Stéphane Jaouen Marc Wolff and Lise-Marie Imbert-Gérard. Conservative numerical methods for a two-temperature resistive MHD model with self-generated magnetic field term. In *CEMRACS'10 research achievements: Numerical modeling of fusion*, volume 32. 2011. pages 158

[Nam01] Raymond Namyst. Habilitation à diriger les recherches. 2001. pages 23

[NAR90] H Penny Nii, Nelleke Aiello, and James Rice. *Experiments on Cage and Poligon: Measuring the performance of parallel blackboard systems*. Morgan Kaufmann Publishers Inc., 1990. pages 73

[NAW+96] W. E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe, and K. Solchenbach. VAMPIR : Visualization and Analysis of MPI Resources. *Supercomputer*, 1996. pages 172

[NE01] Neophytos Neophytou and Paraskevas Evripidou. Net-dbx: a web-based debugger of MPI programs over low-bandwidth lines. *Parallel and Distributed Systems, IEEE Transactions on*, 12(9):986–995, 2001. pages 62

[NET13] NetCDF : Network Common Data Form. http://www.unidata.ucar.edu/software/netcdf/, 2013. pages 74

[NFAR88] H. Penny Nii, Edward A. Feigenbaum, John J. Anton, and A. J. Rockmore. Readings from the AI magazine. chapter Signal-to-symbol transformation: HASP/SIAP case study. 1988. pages 72

[NL91] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991. pages 138

[NMM+08] Aroon Nataraj, Allen D Malony, Alan Morris, Dorian Arnold, and Barton Miller. A framework for scalable, parallel performance monitoring using tau and mrnet. In *International Workshop on Scalable Tools for High-End Computing (STHEC 2008), Island of Kos, Greece*, 2008. pages 66

[NS07a] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74. ACM, 2007. pages 68

[NS07b] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007. pages 64, 68

[Nvi11] CUDA Nvidia. NVIDIA CUDA programming guide, 2011. pages 23

[oGC02] Great Britain. Office of Government Commerce. *Managing Successful Projects with PRINCE2*. Prince Guidance Series. H.M. Stationery Office, 2002. pages 32

[Ope11] OpenACC. The OpenACC™ Application Programming Interface. http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, 2011. pages 24

[Ora07] Orace. Sun Studio 12: Thread Analyzer User's Guide. http://docs.oracle.com/cd/E19205-01/820-0619/820-0619.pdf, 2007. pages 68

[PBAL09] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009. pages 23

[PCDJ12] Marc Pérache, Patrick Carribault, Francois Diakhate, and Hervé Jourdren. MPC: A unified parallel framework for HPC. In *CHOCS (Numéro 41), Revue Scientifique et Technique de la Direction des applications militaires*, pages 22–29. CEA,DAM, 2012. pages 14

[PCJ09] Marc Pérache, Patrick Carribault, and Hervé Jourdren. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In *PVM/MPI*, pages 94–103, 2009. pages 14

[PCJ10] Marc Pérache, Patrick Carribault, and Hervé Jourdren. User level DB: a debugging API for user-level thread libraries. In *IPDPS Workshops*, pages 1–7, 2010. pages 14

[PD11] Steven J Plimpton and Karen D Devine. MapReduce in MPI for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011. pages 75

[Per03] Bruce Perens. Electric Fence malloc() Debugger. http://perens.com/FreeSoftware/ElectricFence/, 2003. pages 68

[Pér06] Marc Pérache. *Contribution à l'élaboration d'environnements de programmation dédiés au calcul scientifique hautes performances*. Thèse de doctorat, spécialité informatique, CEA/DAM Île de France, Université de Bordeaux 1, Domaine Universitaire, 351 Cours de la libération, 33405 Talence Cedex, October 2006. 141 pages. pages 14

[Pfi01] Gregory F Pfister. An introduction to the InfiniBand architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001. pages 25

[PGK+07] Salman Pervez, Ganesh Gopalakrishnan, Robert M Kirby, Robert Palmer, Rajeev Thakur, and William Gropp. Practical model-checking method for verifying correctness of mpi programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 344–353. Springer, 2007. pages 67

[PJN08] Marc Pérache, Hervé Jourdren, and Raymond Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Euro-Par*, pages 78–88, 2008. pages 14, 23

[PLCG95] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. *WoTUG-18*, pages 17–31, 1995. pages 64

[PP03] M. Poppendieck and T. Poppendieck. *Lean software development: an agile toolkit*. The Agile Software Development Series. Addison-Wesley, 2003. pages 34

[Pro13a] GNU Project. GCC 4.7 Release Series. `http://gcc.gnu.org/gcc-4.7/`, 2013. pages 199

[Pro13b] GNU Project. GDB: The GNU Project Debugger. `https://www.gnu.org/software/gdb/`, 2013. pages 61, 62

[Pro13c] The LLDB Project. The LLDB Debugger. `http://lldb.llvm.org/`, 2013. pages 61, 62

[PS03] Paul Petersen and Sanjiv Shah. OpenMP support in the Intel® thread checker. In *OpenMP Shared Memory Parallel Programming*, pages 1–12. Springer, 2003. pages 68

[RAM03] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 21–, New York, NY, USA, 2003. ACM. pages 61, 62, 65, 66, 75

[RBR+07] Giridhar Ravipati, Andrew R Bernat, Nate Rosenblum, Barton P Miller, and Jeffrey K Hollingsworth. Toward the deconstruction of Dyninst. Technical report, Technical Report, Computer Sciences Department, University of Wisconsin, Madison (ftp://ftp. cs. wisc. edu/paradyn/papers/Ravipati07Symta bAPI. pdf), 2007. pages 64

[Rei93] Steven P Reiss. Trace-based debugging. In *Automated and Algorithmic Debugging*, pages 305–314. Springer, 1993. pages 61

[RN10] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Pearson Education/Prentice Hall, 2010. pages 72

[Rot05] Philip Charles Roth. *Scalable on-line automated performance diagnosis*. PhD thesis, University of Wisconsin, 2005. pages 62

[Rot11] V.M. Rota. *Gestion de projet agile: Avec Scrum, Lean, eXtreme Programming...* Architecte logiciel. Eyrolles, 2011. pages 31, 33

[Rou75] Ph Roussel. *PROLOG: Manuel de Reference et d'Utilisation*. Université d'Aix-Marseille II, 1975. pages 72

[Roy70] Winston W Royce. Managing the development of large software systems. In *proceedings of IEEE WESCON*, volume 26. Los Angeles, 1970. pages 31

[RT00] Robert B Ross and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *in Proceedings of the 4th Annual Linux Showcase and Conference*, pages 391–430, 2000. pages 73

[SB08] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Series in agile software development. Pearson Education International, 2008. pages 34

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, volume 12, 2012. pages 68

[Sch86] Eric Schoen. *The CAOS system*. Department of Computer Science, Stanford University, 1986. pages 73

[SCOJ13] Anthony Scemama, Michel Caffarel, Emmanuel Oseret, and William Jalby. Quantum Monte Carlo for large chemical systems: Implementing efficient strategies for petascale platforms and beyond. *Journal of computational chemistry*, 2013. pages 66

[SdS07] Martin Schulz and Bronis R. de Supinski. PNMPI tools: a whole lot greater than the sum of their parts. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007. pages 67, 125, 126

[SF12] Balaji Subramaniam and Wu-chun Feng. The Green Index: A Metric for Evaluating System-Wide Energy Efficiency in HPC Systems. In *8th IEEE Workshop on High-Performance, Power-Aware Computing (HPPAC)*, Shanghai, China, May 2012. pages 24

[SGS+11] Zoltán Szebenyi, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, Felix Wolf, and Brian J. N. Wylie. Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs. In *IPDPS, Anchorage, AK, USA*. IEEE Computer Society, 2011. pages 65, 172

[SH02] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244, 2002. pages 73

[Sho76] Edward Hance Shortliffe. *Computer-based medical consultations: MYCIN*, volume 388. Elsevier New York, 1976. pages 72

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM, 2009. pages 68

[Sim74] Herbert A Simon. The structure of ill structured problems. *Artificial intelligence*, 4(3):181–201, 1974. pages 177

[Sim97] H.A. Simon. *Administrative Behavior, 4th Edition*. Free Press, 1997. pages 30, 35, 36, 37, 177

[SKMP04] Stephan Seidl, A Knüpfer, and R Müller-Pfefferkorn. VTF3-A Fast Vampir Trace File Low-Level Management Library. Technical report, Technical report, ZIH TU Dresden, 2004. pages 64

[SM93] Sekhar R. Sarukkai and Allen D. Malony. Perturbation Analysis of High Level Instrumentation for SPMD Programs. In *PPOPP*, pages 44–53, 1993. pages 71

[SMH98] Sameer Shende, Allen D Malony, and Steven T Hackstadt. Dynamic performance callstack sampling: Merging TAU and DAQV. In *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, pages 515–520. Springer, 1998. pages 65

[SML+12] Wyatt Spear, Allen D Malony, Chee Wai Lee, Scott Biersdorff, and Sameer Shende. An approach to creating performance visualizations in a parallel profile analysis tool. In *Euro-Par 2011: Parallel Processing Workshops*, pages 156–165. Springer, 2012. pages 66

[SMS99] Timothy J Sheehan, Allen D Malony, and Sameer S Shende. A runtime monitoring framework for the tau profiling system. In *Computing in Object-Oriented Parallel Environments*, pages 170–181. Springer, 1999. pages 66

[Sof13] Rogue Wave Software. Totalview. http://www.roguewave.com/products/totalview.aspx, 2013. pages 62

[SS91] Chia-Shiang Shih and John A Stankovic. Survey of deadlock detection in distributed concurrent programming environments and its application to real-time systems and Ada. Technical report, Citeseer, 1991. pages 156

[SSE90] Leon Sterling, Ehud Shapiro, and Michel Eytan. *The art of Prolog*. Wiley Online Library, 1990. pages 72

[Ste90] Guy L Steele. *Common LISP: the language*. Digital Pr, 1990. pages 72

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005. pages 20

[SW04] Fengguang Song and Felix Wolf. CUBE User Manual. Technical Report ICL-UT-04-01, University of Tennessee, Innovative Computing Laboratory, 2004. pages 64, 65

[SWW11] Zoltán Szebenyi, Felix Wolf, and Brian JN Wylie. Performance Analysis of Long-running Applications. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 2105–2108. IEEE, 2011. pages 65

[Sze12] Zoltán Szebenyi. *Capturing Parallel Performance Dynamics*. PhD thesis, RWTH Aachen University, volume 12 of IAS Series, Forschungszentrum Jülich, 2012. ISBN 978-3-89336-798-6. pages 65

[TAMC10] Nathan R Tallent, Laksono Adhianto, and John M Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010. pages 67

[TCP12] Marc Tchiboukdjian, Patrick Carribault, and Marc Pérache. Hierarchical Local Storage: Exploiting Flexible User-Data Sharing Between MPI Tasks. In *IPDPS*, pages 366–377, 2012. pages 14

[Tea06] LAM/MPI Team. XMPI – A Run/Debug GUI for MPI. http://www.lam-mpi.org/software/xmpi/, 2006. pages 61

[Tea13a] Redis Team. Redis, open source, BSD licensed, advanced key-value store. http://redis.io/, 2013. pages 74

[Tea13b] Voldemort Team. Project Voldemort A distributed database. http://www.project-voldemort.com, 2013. pages 74

[The13] The HDF Group. Hierarchical data format version 5. http://www.hdfgroup.org/HDF5, 2013. pages 74

[TLG97] Rajeev Thakur, Ewing Lusk, and William Gropp. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical report, Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 1997. pages 74

[TMC09] Nathan R Tallent and John M Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *ACM Sigplan Notices*, volume 44, pages 229–240. ACM, 2009. pages 67

[TMCP10] Nathan R Tallent, John M Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *ACM Sigplan Notices*, volume 45, pages 269–280. ACM, 2010. pages 67

[TN86] Hirotaka Takeuchi and Ikujiro Nonaka. The New New Product Development Game. *Harvard Business Review*, 1986. pages 34

[TOP10] TOP 500. Tera 100 Supercomputer. http://top500.org/system/10589, 2010. pages 24

[TOP12] TOP 500. Curie Supercomputer (thin nodes). http://top500.org/system/177818, 2012. pages 24, 172

[Uni07] United States Department of Transportation. Systems Engineering for Intelligent Transportation Systems. http://ops.fhwa.dot.gov/publications/seitsguide/seguide.pdf, 2007. pages 32

[VCR93] Paulo Veríssimo, Antonio Casimiro, and Luís Rodrigues. Using Atomic Broadcast to Implement a posteriori Agreement for Clock Synchronization. In *SRDS*, pages 115–124, 1993. pages 70

[VdS00] Jeffrey S Vetter and Bronis R de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 51–51. IEEE, 2000. pages 67

[Vet13] J.S. Vetter. *Contemporary High Performance Computing: From Petascale Toward Exascale*. Chapman and Hall/CRC Computational Science Series. Taylor & Francis Group, 2013. pages 24, 26, 74, 84

[VM01] Jeffrey S. Vetter and Michael O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, 2001. pages 67

[VRC97] Paulo Veríssimo, Luís Rodrigues, and Antonio Casimiro. CesiumSpray: a Precise and Accurate Global Time Service for Large-scale Systems. *Real-Time Systems*, 12(3):243–294, 1997. pages 70

[WB04] Felix Wolf and Nikhil Bhatia. EARL-API Documentation. Technical report, Technical Report ICL-UT-04-03, University of Tennessee, Innovative Computing Laboratory, 2004. pages 64

[WBS+00] C Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 50–50. IEEE, 2000. pages 64

[Wei08] Josef Weidendorfer. Sequential performance analysis with callgrind and kcachegrind. In *Tools for High Performance Computing*, pages 93–113. Springer, 2008. pages 64

[WGM+10] Brian J. N. Wylie, Markus Geimer, Bernd Mohr, David Böhme, Zoltán Szebenyi, and Felix Wolf. Large-scale performance analysis of Sweep3D with the Scalasca toolset. *Parallel Processing Letters*, 20(4):397–414, December 2010. pages 65

[Whi12] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012. pages 75

[Wie61] N. Wiener. *Cybernetics: Or, Control and Communication in the Animal and the Machine*. The @MIT paperback series: Massachusetts Institute of Technology. Mit Press, 1961. pages 35, 36

[Wil88] Mark Williams. Hierarchical Multi-expert Signal Understanding,". *Blackboard Systems, Robert Engelmore and Tony Morgan, editors, Addison-Wesley*, pages 387–415, 1988. pages 73

[WJIG11] Marc Wolff, Stéphane Jaouen, and Lise-Marie Imbert-Gérard. Conservative numerical methods for a two-temperature resistive MHD model with self-generated magnetic field term. In *CEMRACS'10 research achievements: Numerical modeling of fusion*. 2011. pages 153, 171

[WJR07] J.P. Womack, D.T. Jones, and D. Roos. *The Machine That Changed the World: The Story of Lean Production– Toyota's Secret Weapon in the Global Car Wars That Is Now Revolutionizing World Industry*. Free Press, 2007. pages 34

[WK82] James Q. Wilson and George L. Kelling. The police and neighborhood safety: Broken windows, 1982. pages 49

[WM03] Felix Wolf and Bernd Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. In *Proc. of 11th Euromicro Workshop on Parallel Distributed and Network-Based Processing (PDP), Genua, Italy*, pages 13–22. IEEE Computer Society, February 2003. pages 64

[WM04] Felix Wolf and Bernd Mohr. *EPILOG binary trace-data format*. FZJ-ZAM, 2004. pages 64, 112

[Wol03] Felix Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. PhD thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003. ISBN 3-00-010003-2. pages 64

[Wol11] Marc Wolff. *Analyse mathématique et numérique du système de la magnétohydrodynamique résistive avec termes de champ magnétique autogénéré*. PhD thesis, Université de Strasbourg, 2011. pages 153

[WR07] H'sien Jin Wong and Alistair P Rendell. The design of MPI based distributed shared memory systems to support OpenMP on clusters. In *Cluster Computing, 2007 IEEE International Conference on*, pages 231–240. IEEE, 2007. pages 139

[ws13] Amazon web services. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/en/s3/, 2013. pages 74

[Zel09] A. Zeller. *Why programs fail: a guide to systematic debugging*. Morgan Kaufman Incorporated, 2009. pages 47, 48

[ZLG+99] Omer Zaki, Ewing Lusk, William Gropp, Deborah Swider, et al. Toward scalable performance visualization with Jumpshot. *International Journal of High Performance Computing Applications*, 13:277–288, 1999. pages 64

Appendices

APPENDIX A

Instrumentation Filtering at Compiler-Level

This section presents some work which has been done on the GCC compiler in order to filter instrumented functions in a more effective way. After presenting existing filtering possibilities and their limitations, we present our patch which extends filtering possibilities. Eventually we perform some performance measurements to illustrate its use and demonstrate the performance gains it provides.

A.1 Existing Filtering

Command	Effect
-finstrument-functions-exclude-file-list=file,...	Filter a list of file from which functions wont be instrumented. **The match is done on substrings**.
-finstrument-functions-exclude-function-list=sym,...	Filter a list of symbols which wont be instrumented. **The match is done on substrings**.

Figure A.1: *Description of the two filtering options associated with the -finstrument-functions flag.*

By looking at the GCC (4.7.1) documentation [Pro13a] which is summed up in figure A.1, it can be seen that "finstrumented-functions" can be filtered using two command line options which operate either on the source file or on symbol name. In particular, the filter is applied if it is a substring of a given file or symbol. Although practical, for example, to filter files from a given directory or symbol from a single library (with same prefix), this approach makes selective instrumentation not practical. Indeed, substrings matching can lead to involuntary filtering of function even when providing exact names, possibility which becomes a highly probable when handling large simulation programs (thousands of functions). Moreover, when filtering in the purpose of reducing the overhead, it would be easier to white-list instead of blacklisting them as the target set is by definition smaller, feature not supported by current interface. The way lists are passed can also be problematic when managing large sets as command lines cannot be too large without quickly becoming unmanageable, an alternative way of passing them could therefore be useful to leverage this limitation.

A.2 Proposed Extension

This section details the functionalities added by our $\approx$ 270 lines patch to GCC 4.7.1 which aimed at fixing the limitations we identified in previous section. This small patch modified three files gcc/opts.c, gcc/gimplify.c and gcc/common.opt and left previous flags unmodified to ensure compatibility. Our purpose was to fix the following limitations of GCC's instrumentation filtering:

- Filtering matches are only done on substrings (current implementation calls substr).
- There is no white-listing mechanism.
- Filter list can only be passed as a command line argument.

In this purpose we extended the existing filtering commands with eight new ones which support either substring or exact matching, allow white-listing and blacklisting and which are able to retrieve the list either from the command line or from a file.

Command	Effect
-finstrument-functions-blacklist-function-list=sym,...	Blacklists a list of symbols.
-finstrument-functions-blacklist-file-list=file,...	Blacklists a list of files.
-finstrument-functions-blacklist-function-from-file=sym.txt	Blacklists a list of symbols taken from a file.
-finstrument-functions-blacklist-file-from-file=file.txt	Blacklists a list of files taken from a file.
-finstrument-functions-whitelist-function-list=sym,...	White-lists a list of symbols.
-finstrument-functions-whitelist-file-list=file,...	White-lists a list of files.
-finstrument-functions-whitelist-function-from-file=sym.txt	White-lists a list of symbols taken from a file.
-finstrument-functions-whitelist-file-from-file=file.txt	White-lists a list of files taken from a file.

Figure A.2: *Additions to filtering options associated with the -finstrument-functions flag.*

Figure A.2 presents the eight new filtering options which are added by the patch. Note that the two first ones provide are similar to previously existing functions. However, it shall be noted that they still provide an extra feature. Indeed, in order to propose both sub-string matching and exact matching we adopted the convention of preceding symbol name with an '=' to perform an exact match.

APPENDIX B

Instrumenting the MPC Framework

This appendix presents some point which might help tool developer who would like to instrument the MPC runtime. We start by describing the extended TLS mechanism of MPC, then we present some useful instrumentation points. Eventually, we detail all the topology getters which can be used to provide a context to the instrumented events.

B.1 MPC Extended TLS

As presented in section 1.1, MPC executes "MPI tasks" in user level threads, making the use of classical Thread Local Storage (TLS) impossible as they would be shared between multiple tasks, therefore, failing a providing an unique context to the instrumentation. However, thanks to the extended TLS mechanism [CPJ11], MPC can switch those values at scheduler level in order to maintain a context for each task. Their default value is NULL. As our instrumentation library required such TLS, two of them were added to the runtime. To use them just declare two extern variables:

- extern __thread void *tls_trace_module;
- extern __thread void *tls_args;

These variables are declared in sctk_tls.h and can be used to store the instrumentation context in post initialisation code (see next section).

B.2 Launch Hooks

As developed in section 9.3.3, two hooks have been added to capture MPC's initialisation at both process and task level, allowing the setup of respectively global and local contexts.

- **void MPC_Process_hook()**
  - Called once for each process.
  - You cannot do MPI calls from here.
  - MPC TLS are *not* available.
- **void MPC_Task_hook(int rank)**
  - Called once for each MPI task.
  - You can do MPI calls from here.
  - MPC TLS are available.

B.3 Instrumentation Points

This section details common instrumentation points which can be used to retrieve useful events from the MPC runtime. After presenting the profiling interface, we provide examples on how to intercept thread creation and lock related calls.

B.3.1 MPI Profiling Interface

MPC implements PMPI and PMPC calls they are both the same except that PMPI is called by MPI_* calls (classical MPI code) and PMPC is called by MPC_* by explicitly MPC codes. If you compile codes including <mpi.h> with mpc_cc they will be automatically redirected to MPC. PMPC calls were not redirected to PMPI calls because they slightly extend the MPI api.

B.3.2 Thread Spawning

As the TLS are only available after the creation of the thread you have to use MPC_Task_hook to set them up these tasks are created with sctk_thread_create in sctk_thread.c:685. Further thread creation are done by sctk_thread.c:784. Note that those symbol have no associated weak symbol because we currently intercept it by preloading our library as presented in Figure B.1.* Note that MPC creates a number of worker threads which should be ignored or at least processed separately. You should instrument only threads which were created in a parent which TLS was setup by MPC_Task_hook.

```
#include <mpc.h>

int sctk_thread_create (sctk_thread_t * thread,
        const sctk_thread_attr_t * attr,
        void *(*start_routine) (void *),
        void *arg, long task_id)
{
    int tmp = 0;
    tmp = _sctk_thread_create( thread, attr, THREAD_HOOK,  THREAD_STRUCT, task_id);
    return tmp;
}

int sctk_user_thread_create (sctk_thread_t * thread,
        const sctk_thread_attr_t * attr,
        void *(*start_routine) (void *),
        void *arg)
{
    int tmp = 0;
    tmp = _sctk_user_thread_create ( thread, attr, THREAD_HOOK,  THREAD_STRUCT);
   return tmp;
}
```

Figure B.1: *Illustration of the interception of MPC threads creation.*

B.3.3 Lock Instrumentation

Mutex_lock and mutex_unlock have their weak symbols, allowing them to be instrumented as in Figure B.2.

```
int user_sctk_thread_mutex_lock (sctk_thread_mutex_t * mutex)
{
    int ret =  0;
   ret = sctk_thread_mutex_lock (mutex);
   return ret;
}
int user_sctk_thread_mutex_unlock (sctk_thread_mutex_t * mutex)
{
    int ret = 0;
    ret = sctk_thread_mutex_unlock (mutex);
    return ret;
}
```

Figure B.2: *Sample instrumentation of MPC locks.*

B.4 Topology Getters

MPC is hierarchical : one node hosts one or multiple processes which embed VCPUS (user-space abstraction of cores) which themselves are used to run MPC tasks. Once MPC is initialised (MPC_Task_hook) it is possible to retrieve the various identifiers as presented in figure B.3 and use them as in figure B.4.

Description	MPC Call
Node rank	MPC_Node_rank(int *n)
Node count	MPC_Node_number(int *n)
VCPU Rank	MPC_Processor_rank(int *n)
VCPU count	MPC_Processor_number(int *n)
Local process rank (on node)	MPC_Local_process_rank(int *n)
Local process count	MPC_Local_process_number(int *n)
Process rank (global)	MPC_process_rank(int *n)
Process count (global)	MPC_process_number(int *n)
MPI task rank	MPC_Comm_rank(MPC_COMM_WORLD, int *n)
MPI task count	MPC_Comm_size(MPC_COMM_WORLD, int *n)
Thread id and task rank	sctk_get_thread_info (int *task_id, int *thread_id) (Note that this thread id is not unique)

Figure B.3: *List of topology getters available in MPC.*

```c
#include <stdio.h>
#include <mpc.h>
#include <pthread.h>

void *foo( void *a ) {
  int thread_id,  task_id;
  sctk_get_thread_info (&task_id, &thread_id);
  printf("CThid : %d \t CTskid : %d \n", thread_id, task_id);
}

int main(int argc, const char *argv[]) {
    int node, node_c, proc, proc_c, local, local_c, process;
    int process_c, rank, size, thread_id,  task_id;

    sctk_get_thread_info (&task_id, &thread_id);
    MPC_Node_rank( &node );
    MPC_Node_number( &node_c );
    MPC_Processor_rank( &proc );
    MPC_Processor_number( &proc_c );
    MPC_Local_process_rank( &local );
    MPC_Local_process_number( &local_c );
    MPC_Process_rank( &process );
    MPC_Process_number( &process_c );
    MPC_Comm_rank( MPC_COMM_WORLD, &rank);
    MPC_Comm_size( MPC_COMM_WORLD, &size);

    pthread_t th;
    pthread_create( &th, NULL, foo, NULL );
    int i = 0;
    for( i = 0 ; i < size; i++ ) {
        if( i == rank ) {
            printf("========================\n");
            printf("R : %d \t RC : %d\n", rank, size);
            printf("Thid : %d \t Tskid : %d \n", thread_id, task_id);
            printf("N : %d \t NC : %d\n", node, node_c);
            printf("PROC : %d \t PROCC : %d\n", proc, proc_c);
            printf("LP : %d \t LPC : %d\n", local, local_c);
            printf("PR : %d \t PC : %d\n", process, process_c);
            printf("========================\n");
        }
        MPC_Barrier( MPC_COMM_WORLD );
    }
    pthread_join( th, NULL );
    return 0;
}
```

Figure B.4: *Example using every topology getters in MPC.*